Module 4

SQL- Advances Queries

1.1 More Complex SQL Retrieval Queries

Additional features allow users to specify more complex retrievals from database

1.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value

Example

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 CollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

Each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL 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

Table 5.1 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

The rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query.

In select-project-join 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 using the comparison operators **IS** or **IS NOT.**

Example: Retrieve the names of all employees who do not have supervisors.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL;
```

1.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**

Example1: List the project numbers of projects that have an employee with last name 'Smith' as manager

```
SELECT DISTINCT Pnumber FROM PROJECT WHERE
 Pnumber IN
(SELECT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE
 WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='smith');
```

Example2: List the project numbers of projects that have an employee with last name 'Smith' as either manager or as worker.

```
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');
```

We make use of comparison operator **IN**, which compares a value *v* with a set (or multiset) of values *V* and evaluates to **TRUE** if *v* is one of the elements in *V*.

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. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. For example, the following 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' );
```

In this example, the IN operator compares the subtuple of values in parentheses (Pno,Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

Nested Queries::Comparison Operators

Other comparison operators can be used to compare a single value v to a set or multiset *V.* 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. 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*. For 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 );
```

In general, we can have several levels of nested queries. We can once again be faced with possible 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**.

To avoid potential errors and ambiguities, create tuple variables (aliases) for all tables referenced in SQL query

Example: Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee

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

In the above nested query, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex.

1.1.3 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**.

Example:

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

The nested query is evaluated once for each tuple (or combination of tuples) in the outer query. we can think of query in above example as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

1.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS Functions

The EXISTS function in SQL is used to check whether the result of a correlated 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.

For example, the query to retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee can be written using EXISTS functions as follows:

```
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);
```

Example: List the names of managers who have at least one dependent

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

In general, EXISTS(Q) returns **TRUE** if there is at least one tuple in the result of the nested query Q, and it returns **FALSE** otherwise.

NOT EXISTS Functions

NOT EXISTS(Q) returns **TRUE** if there are no tuples in the result of nested query Q, and it returns **FALSE** otherwise.

Example: Retrieve the names of employees who have no dependents.

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

For each EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

Example: Retrieve the name of each employee who works on all the projects controlled by department number 5

```
SELECT Fname, Lname

FROM EMPLOYEE
WHERE NOT EXISTS ( ( SELECT Pnumber
FROM PROJECT
WHERE Dnum=5)
EXCEPT ( SELECT Pno
FROM WORKS_ON
WHERE Ssn=Essn) );
```

UNIQUE Functions

UNIQUE(Q) returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

1.1.5 Explicit Sets and Renaming of Attributes in SQL

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

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

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

In SQL, 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

Example: Retrieve the last name of each employee and his or her supervisor

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

1.1.6 Joined Tables in SQL and Outer Joins

An SQL join clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two tables by using values common to each. SQL specifies four types of JOIN

1. INNER,
2. OUTER
3. EQUIJOIN and
4. NATURAL JOIN

INNER JOIN

An inner join is the most common join operation used in applications and can be regarded as the default join-type. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join- predicate (the condition). The result of the join can be defined as the outcome of first taking the Cartesian product (or Cross join) of all records in the tables (combining every record in table A with every record in table B)—then return all records which satisfy the join predicate

Example: **SELECT** * **FROM** employee

INNER JOIN department **ON**

employee.dno = department.dnumber;

EQUIJOIN and NATURAL JOIN

An **EQUIJOIN** is a specific type of comparator-based join that uses only equality comparisons in the join-predicate. Using other comparison operators (such as <) disqualifies a join as an equijoin.

NATURAL JOIN is a type of EQUIJOIN where the join predicate arises implicitly by comparing all columns in both tables that have the same column-names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns.

```
SELECT    Fname, Lname, Address
FROM      EMPLOYEE NATURAL JOIN
          DEPARTMENT
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. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause.

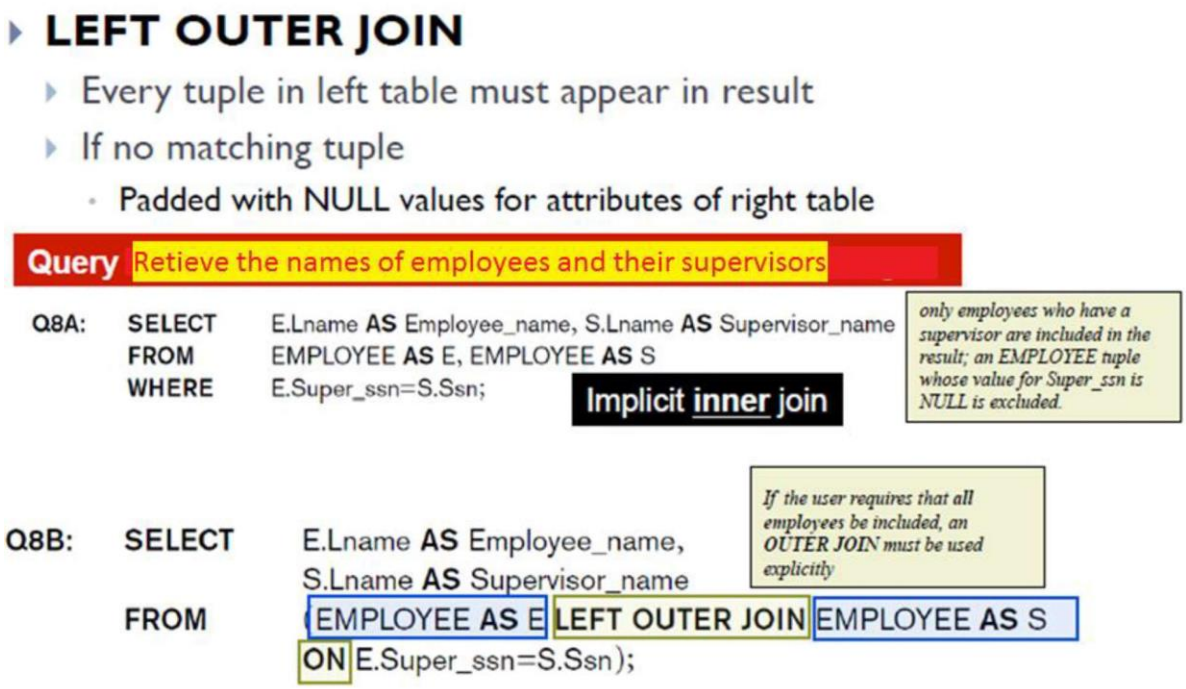
CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.

OUTER JOIN

An outer join does not require each record in the two joined tables to have a matching record. The joined table retains each record-even if no other matching record exists**.** Outer joins subdivide further into

- Left outer joins
- Right outer joins
- Full outer joins

No implicit join-notation for outer joins exists in standard SQL.



- RIGHT OUTER jOIN

  Evei y tuple in right table must appear in result

  If no matching tuple

  Padded with NULL values for the attributes of lek table

- FULL OUTER jOIN

  a full outer join combines the effect of applying both left and i ight outer joins.

  Where recoi ds in the FULL OUTER jOlNed tables do not match, the result set will have NULL values foi every column of the table that lacks a matching row.

  For those recoi ds that do match, a single i ow will be produced in the result set (containing fields populated fi om both tables).

- Not all SC,jL implementations have implemented the new syntax of joined tables.
- In some systems, a different syntax wzs used to specify outer joins by using the comparison operators +=, =+, and +=+ for lek, right, and full outer join, respectively
- For example, this synt»x is available in Oracle.To specify the left outer join in 8B using this syntax, we could write the query Q8C as follows:

```
Q8C:    SELECT    E.Lname, S.Lname
        FROM      EMPLOYEE E, EMPLOYEE S
        WHERE     E.Super_ssn += S.Ssn;
```

MULTIWAY JOIN

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**.

Example: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

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

1.1.7 Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. 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 and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECTclause or in a HAVING clause (which we introduce later). 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.

Examples

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

```
SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM EMPLOYEE;
```

2. 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.

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

3. Count the number of distinct salary values in the database.

```
SELECT COUNT (DISTINCT Salary)
FROM EMPLOYEE;
```

4. To retrieve the names of all employees who have two or more dependents

SELECT Lname, Fname

FROM EMPLOYEE

WHERE (**SELECT COUNT** (*)

FROM DEPENDENT

WHERE Ssn=Essn) >= 2;

1.1.8 Grouping: The GROUP BY and HAVING Clauses

Grouping is used to create subgroups of tuples before summarization. For example, we may want to find the average salary of employees *in each department* or the number of employees who work *on each project*. In these cases we need to **partition** the relation into non overlapping 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)**.

SQL has a **GROUP BY** clause for this purpose. 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).

Example: For each department, retrieve the department number, the number of employees in the department, and their average salary.

SELECT Dno, **COUNT** (*), **AVG** (Salary)

FROM EMPLOYEE

GROUP BY Dno;

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a NULL value in the grouping attribute. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result

of query

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

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

Above query shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations.

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.

Example: 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.

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

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

Example: For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

SELECT Pnumber, Pname, **COUNT** (*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno **AND** Ssn=Essn **AND** Dno=5
GROUP BY Pnumber, Pname;

Example: For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

SELECT Dnumber, **COUNT** (*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno **AND** Salary>40000 **AND**
(**SELECT** Dno
FROM EMPLOYEE
GROUP BY Dno
HAVING COUNT (*) > 5);

1.1.9 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two SELECT and FROM are mandatory.The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way.The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

SELECT <attribute and function list>
FROM <table list>
[**WHERE** <condition>]
[**GROUP BY** <grouping attribute(s)>]
[**HAVING** <group condition>]
[**ORDER BY** <attribute list>];

The **SELECT** clause lists the attributes or functions to be retrieved. The **FROM** clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The **WHERE** clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. **GROUP** BY specifies grouping attributes, whereas **HAVING** specifies a condition on the groups being selected rather than on the individual tuples. Finally, **ORDER BY** specifies an order for displaying the result of a query.

A query is evaluated conceptually by first applying the FROM clause to identify all tables involved in the query or to materialize any joined tables followed by the WHERE clause to select and join tuples, and then by GROUP BY and HAVING. ORDER BY is applied at the end to sort the query result Each DBMS has special query optimization routines to decide on an execution plan that is efficient to execute

In general, there are numerous ways to specify the same query in SQL.This flexibility in specifying queries has advantages and disadvantages.

- The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison. From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.
- The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way

1.2 Specifying Constraints as Assertions and Actions as Triggers

1.2.1 Specifying General Constraints as Assertions in SQL

Assertions are used to specify additional types of constraints outside scope of built-in relational model constraints. In SQL, users can specify general constraints via declarative assertions, using the **CREATE ASSERTION** statement of the DDL.Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

General form :

CREATE ASSERTION <Name_of_assertion> **CHECK** (<cond>)

For the assertion to be satisfied, the condition specified after CHECK clause must return true.

For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for in SQL, we can write the following assertion:

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 refer to the constraint or to modify or drop it. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.

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. Thus, the assertion is violated if the result of the query is not empty

Example: consider the bank database with the following tables

- *branch (branch_name, branch_city, assets)*
- *customer (customer_name, customer_street, customer_city)*
- *account (account_number, branch_name, balance)*
- *loan (loan_number, branch_name, amount)*
- *depositor (customer_name, account_number)*
- *borrower (customer_name, loan_number)*

1. Write an assertion to specify the constraint that the Sum of loans taken by a customer does not exceed 100,000

```
CREATE ASSERTION sumofloans
CHECK (100000> = ALL
SELECT customer_name,sum(amount)
FROM borrower b, loan l
WHERE b.loan_number=l.loan_number
GROUP BY customer_name );
```

2. Write an assertion to specify the constraint that the Number of accounts for each customer in a given branch is at most two

```
CREATE ASSERTION NumAccounts
CHECK (   2 >= ALL
SELECT  customer_name,branch_name, count(*)
FROM      account A , depositor D
WHERE  A.account_number = D.account_number
GROUP BY customer_name, branch_name );
```

1.2.2 Introduction to Triggers in SQL

A trigger is a procedure that runs automatically when a certain event occurs in the DBMS. 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. The CREATE TRIGGER statement is used to implement such actions in SQL.

General form:

CREATE TRIGGER <name>

BEFORE | **AFTER** | <events>

FOR EACH ROW |FOR EACH STATEMENT

WHEN (<condition>)

<action>

A trigger has three components

1. **Event:** When this event happens, the trigger is activated
   - Three event types : Insert, Update, Delete
   - Two triggering times: Before the event
     After the event
2. **Condition (optional):** If the condition is true, the trigger executes, otherwise skipped
3. **Action:** The actions performed by the trigger

When the **Event** occurs and **Condition** is true, execute the **Action**

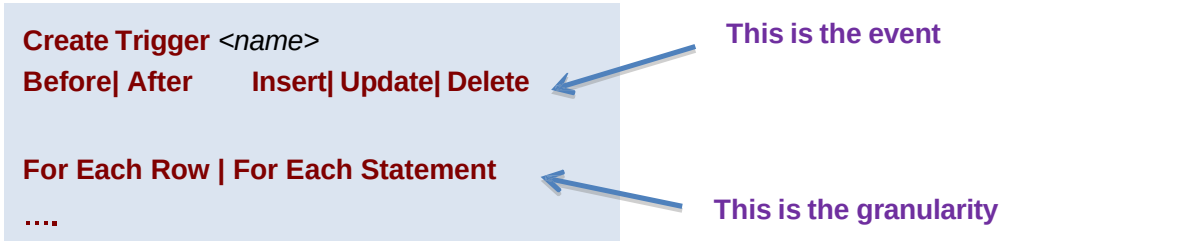
Create Trigger *ABC*
Before Insert On Students

This trigger is activated when an insert statement is issued, but before the new record is inserted

Create Trigger *XYZ*
After Update On Students
….

This trigger is activated when an update statement is issued and after the update is executed

Does the trigger execute for each updated or deleted record, or once for the entire statement ?. We define such granularity as follows:




```
Create Trigger XYZ
After Update ON <tablename>
For each statement
....
```

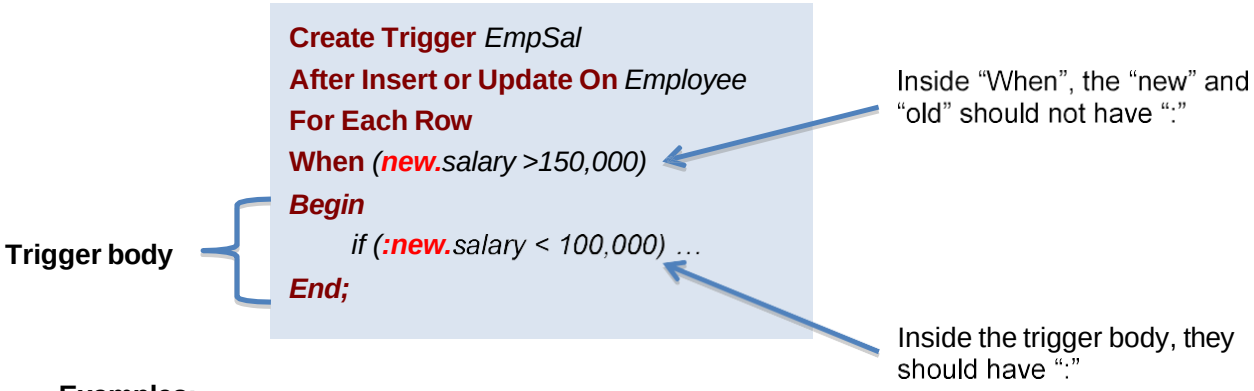
This trigger is activated once (per UPDATE statement) after all records are updated

```
Create Trigger XYZ
Before Delete ON <tablename>
For each row
....
```

This trigger is activated before deleting each record

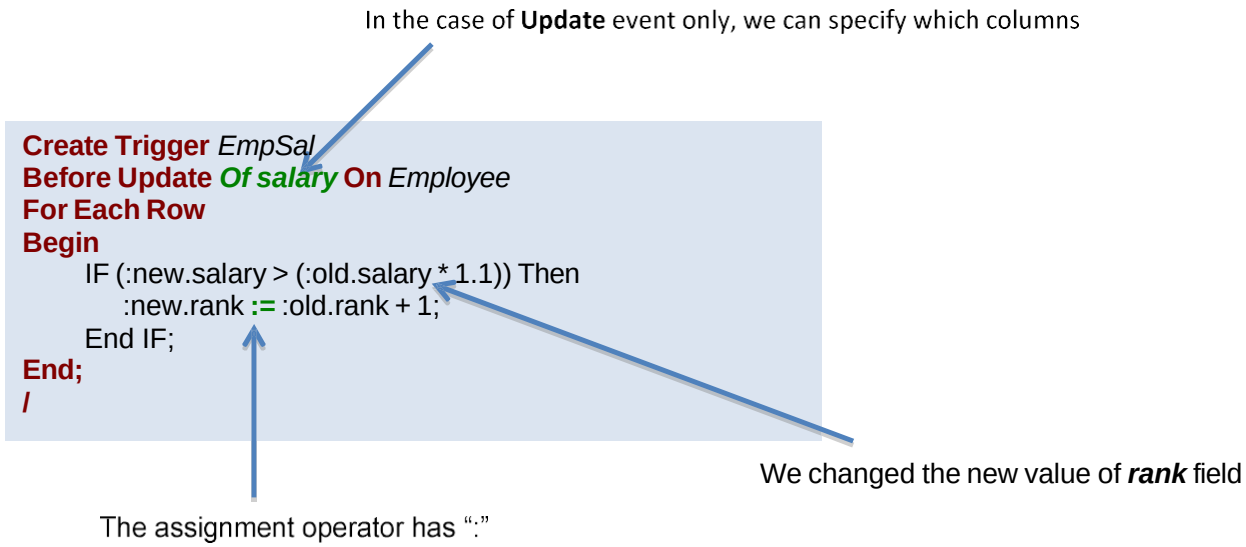
In the action, you may want to reference:

- The new values of inserted or updated records ***(:new)***
- The old values of deleted or updated records ***(:old)***

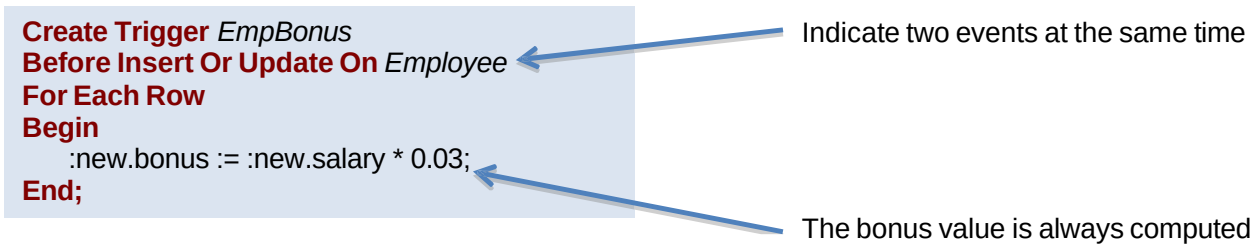


Examples:

1) If the employee salary increased by more than 10%, then increment the rank field by 1.



2) Keep the bonus attribute in Employee table always 3% of the salary attribute



1. 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

```
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 );
```

   - The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later
   - In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor
   - The action is to execute the stored procedure INFORM_SUPERVISOR

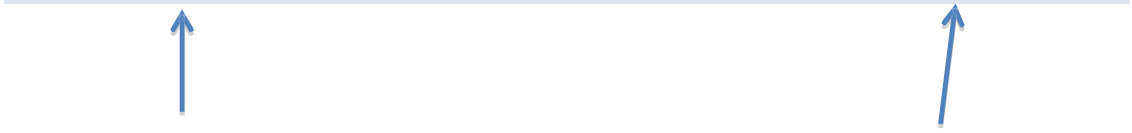
Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates.

Assertions vs. Triggers

- Assertions do not modify the data, they only check certain conditions. Triggers are more powerful because the can check conditions and also modify the data
- Assertions are not linked to specific tables in the database and not linked to specific events. Triggers are linked to specific tables and specific events
- All assertions can be implemented as triggers (one or more). Not all triggers can be implemented as assertions

Example: Trigger vs. Assertion

All new customers opening an account must have opening balance >= $100. However, once the account is opened their balance can fall below that amount.



We need triggers, assertions cannot be used

Trigger Event: Before Insert

```
Create Trigger OpeningBal
Before Insert On Customer
For Each Row
Begin
   IF (:new.balance is null or :new.balance < 100) Then
    RAISE_APPLICATION_ERROR(-20004, 'Balance should be >= $100');
   End IF;
End;
```

1.3 Views (Virtual Tables) in SQL

1.3.1 Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables. other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be 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. 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.

For example, referring to the COMPANY database, we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE,WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE,WORKS_ON, and PROJECT tables the **defining tables** of the view.

1.3.2 Specification of Views in SQL

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.

Example 1:

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

Example 2:

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

In example 1, we did not specify any new attribute names for the view WORKS_ON1. In this case,WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

Example 2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

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 and specify the query as :

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

The same query would require the specification of two joins if specified on the base relations directly. 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 is not realized or materialized at the time of view definition but rather at the time when we specify a query on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date.

If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it. For example : **DROP VIEW** WORKS_ON1;

1.3.3 View Implementation, View Update and Inline Views

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested.

- One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables. For example, the query

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

would be automatically modified to the following query by the DBMS:

```
SELECT Fname, Lname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn=Essn AND Pno=Pnumber
AND Pname='ProductX';
```

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 queries are going to be applied to the same view within a short period of time.

- The second strategy, called **view materialization**, involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date.

Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a materialized view table when a database update is applied to one of the defining base tables.

The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

Updating of views is complicated and can be ambiguous. In general, 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.

To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1: UPDATE WORKS_ON1
     SET Pname = 'ProductY'
     WHERE Lname='Smith' AND Fname='John'
     AND Pname='ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create additional side effects that affect the result of other queries.

For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(a): UPDATE WORKS_ON
     SET Pno= (SELECT Pnumber
     FROM PROJECT
     WHERE Pname='ProductY' )
     WHERE Essn IN ( SELECT Ssn
     FROM EMPLOYEE
     WHERE Lname='Smith' AND Fname='John' )
     AND
     Pno= (SELECT Pnumber
     FROM PROJECT
     WHERE Pname='ProductX' );
```

(b) : UPDATE PROJECT **SET** Pname = 'ProductY'
WHERE Pname = 'ProductX';

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'.

It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view does not make sense because Total_sal is defined to be the sum of the individual employee salaries. This request is shown as UV2:

UV2: UPDATE DEPT_INFO
SET Total_sal=100000
WHERE Dname='Research';

A large number of updates on the underlying base relations can satisfy this view update.

Generally, a view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to more than one update on the underlying base relations, we must have a certain procedure for choosing one of the possible updates as the most likely one.

In summary, we can make the following observations:

- 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.

In SQL, the clause **WITH CHECK OPTION** must be added at the end of the view definition if a view *is to be updated*. This allows the system to check for view updatability and to plan an execution strategy 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**. In this case, the view is defined within the query itself.

1.4 Schema Change Statements in SQL

Schema evolution commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema.

1.4.1 The DROP Command

The DROP command can be used to drop named schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is no longer needed, the DROP SCHEMA command can be used.

There are two drop behavior options: **CASCADE** and **RESTRICT**. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY **CASCADE**;

If the **RESTRICT** option is chosen in place of **CASCADE**, the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, , we can get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT **CASCADE**;

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints (for example, by foreign key definitions in another relation) or views or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

The DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command should be used instead of DROP TABLE.

The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

1.4.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible **alter table actions** include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.

For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema , we can use the command:

ALTER TABLE COMPANY.EMPLOYEE **ADD COLUMN** Job VARCHAR(12);

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple. If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is not allowed in this case.

To drop a column, we must choose either **CASCADE** or **RESTRICT** for drop behavior. If **CASCADE** is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If **RESTRICT** is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column.

For example, the following command removes the attribute Address from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **DROP DEFAULT**;

ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **SET DEFAULT** '333445555';

Alter Table - Alter/Modify Column

To change the data type of a column in a table, use the following syntax:

ALTER TABLE table_name
MODIFY column_name datatype;

For example we can change the data type of the column named "DateOfBirth" from date to year in the "Persons" table using the following SQL statement:

ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

→

-
-

Module 4

Transaction Processing

5.0 Introduction

The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples:

- airline reservations
- banking
- credit card processing, •
  online retail purchasing,
- Stock markets, supermarket checkouts, and many other applications

These systems require high availability and fast response time for hundreds of concurrent users. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

5.1 Objectives

- To study transaction properties
- To study creation of schedule and maintaining schedule equivalence.
- To check whether the given schedule is serailizable or not.
- To study protocols used for locking objects
- Differentiating between 2PL and Strict 2PL

5.2 Introduction to Transaction Processing

5.2.1 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**

Single-User versus Multiuser Systems

- A DBMS is
- **single-user**
  - at most one user at a time can use the system
  - Eg: Personal Computer System
- **multiuser**
  - many users can use the system and hence access the database concurrently
  - Eg: Airline reservation database

- Concurrent access is possible because of **Multiprogramming.** Multiprogramming can be achieved by:
  - interleaved execution
  - Parallel Processing
- **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**, as illustrated in Figure 21.1

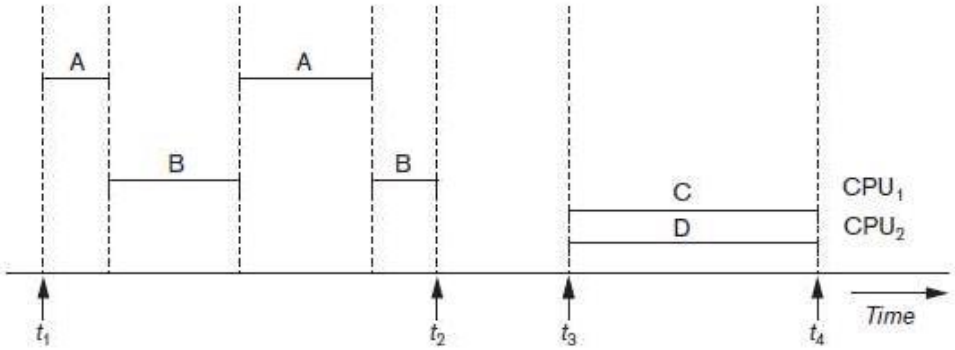


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

- Figure 21.1, shows two processes, A and B, executing concurrently in an interleaved fashion
- Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk
- The CPU is switched to execute another process rather than remaining idle during I/O time
- Interleaving also prevents a long process from delaying other processes.
- If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 21.1
- Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**
- In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

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

A Transaction an executing program that forms a logical unit of database processing

It includes one or more DB access operations such as insertion, deletion, modification or retrieval operation.

It can be either embedded within an application program using **begin transaction** and **end transaction** statements Or specified interactively via a high level query language such as SQL

- Transaction which do not update database are known as **read only transactions.**
- Transaction which do update database are known as **read write transactions.**
- 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, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database
- Each data item has a unique name
- **Basic DB access operations that a transaction can include are:**
  - **read_item(X)**: Reads a DB item named X into a program variable.
  - **write_item(X)**: Writes the value of a program variable into the DB item named X
- **Executing read_item(X) include the following steps:**
  1. Find the address of the disk block that contains item X
  2. Copy the block into a buffer in main memory
  3. Copy the item X from the buffer to program variable named X.
- **Executing write_item(X) include the following steps:**
  1. Find the address of the disk block that contains item X
  2. Copy the disk block into a buffer in main memory
  3. Copy item X from program variable named X into its correct location in buffer.
  4. Store the updated disk block from buffer back to disk (either immediately or later).
- Decision of when to store a modified disk block is handled by **recovery manager** of the DBMS in cooperation with operating system.

  A DB cache includes a number of data buffers.

  When the buffers are all occupied a buffer replacement policy is used to choose one of the buffers to be replaced. EG: LRU

- A transaction includes read_item and write_item operations to access and update DB.

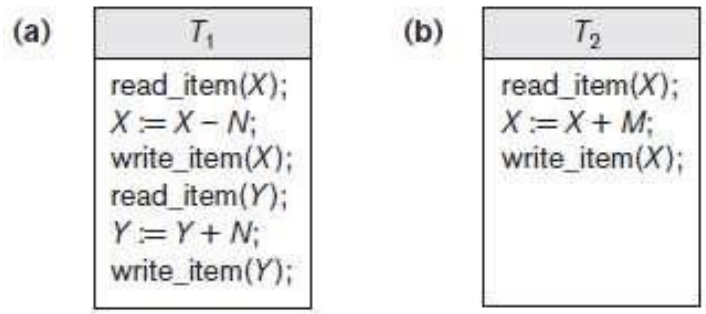
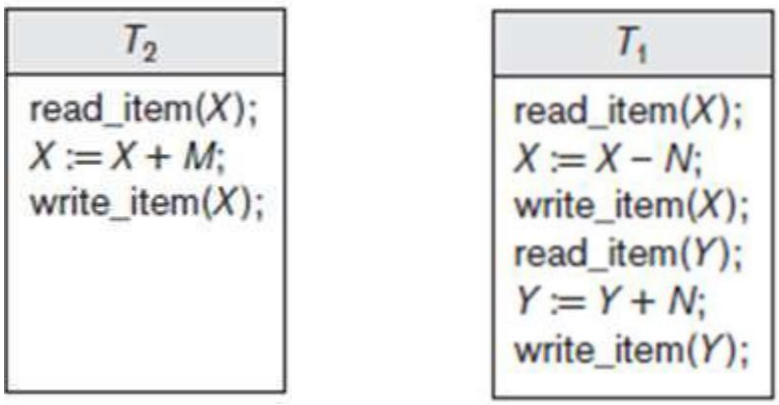


Figure 21.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

- The **read-set** of a transaction is the set of all items that the transaction reads
- The **write-set** is the set of all items that the transaction writes
- For example, the read-set of *T*1 in Figure 21.2 is {*X*, *Y*} and its write-set is also {*X*, *Y*}.

5.2.3 Why Concurrency Control Is Needed

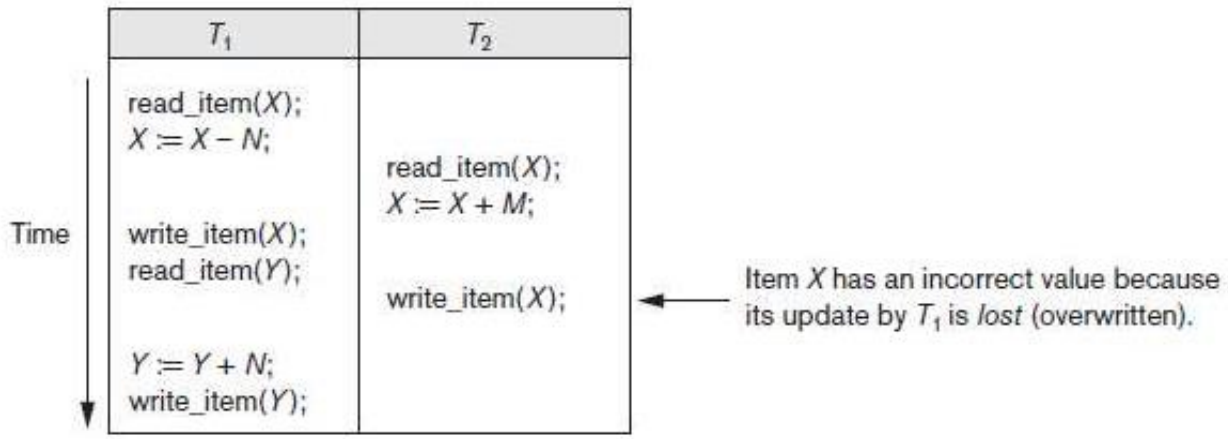
- Several problems can occur when concurrent transactions execute in an uncontrolled manner
- Example:
  - We consider an Airline reservation DB
  - Each records is stored for an airline flight which includes Number of reserved seats among other information.
  - Types of problems we may encounter:
    1. The Lost Update Problem
    2. The Temporary Update (or Dirty Read) Problem
    3. The Incorrect Summary Problem
    4. The Unrepeatable Read Problem



- Transaction T1
  - 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.
- Transaction T2
  - reserves M seats on the first flight (X)

1. The Lost Update Problem

- occurs when two transactions that access the same DB items have their operations interleaved in a way that makes the value of some DB item 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 below



- Final value of item *X* is incorrect because *T*2 reads the value of *X before T*1 changes it in the database, and hence the updated value resulting from *T*1 is lost.

- For example:

  X = 80 at the start (there were 80 reservations on the flight)

  N = 5 (T1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y)

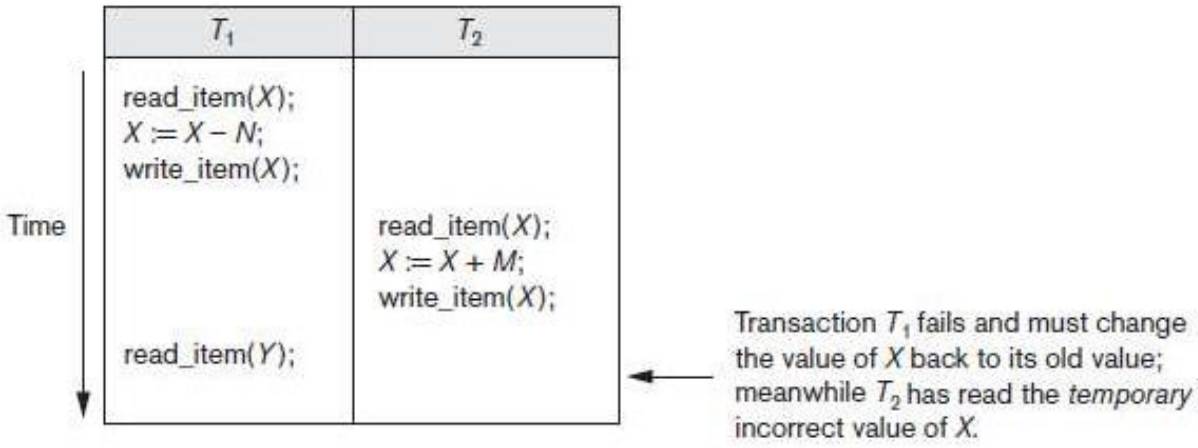
  M = 4 (T2 reserves 4 seats on X)

  The final result should be X = 79.

- The interleaving of operations shown in Figure is X = 84 because the update in T1 that removed the five seats from X was lost.

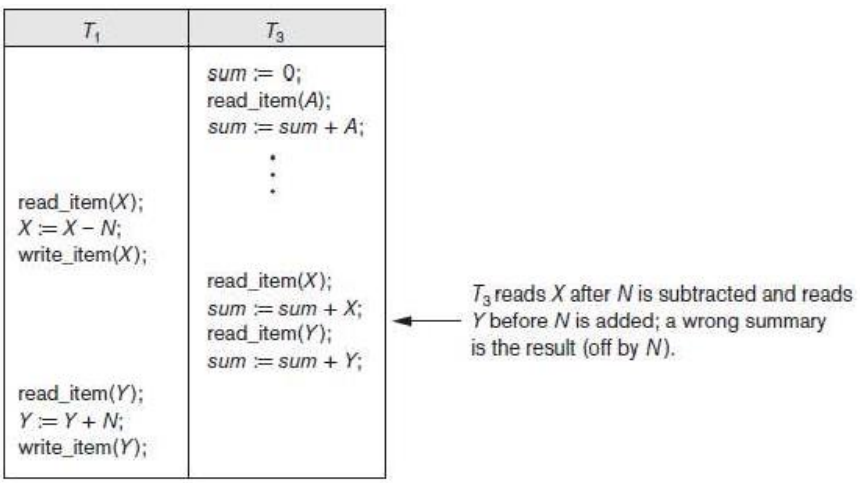
2. The Temporary Update (or Dirty Read) Problem

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



3. The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of db 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.



4. The Unrepeatable Read Problem

- Transaction T reads the same item twice and gets different values on each read, since the item was modified by another transaction T` between the two reads.
- for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights
- When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

5.2.4 Why Recovery Is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either
    1. All the operations in the transaction are completed successfully and their effect is recorded permanently in the database or
    2. The transaction does not have any effect on the database or any other transactions
- In the first case, the transaction is said to be committed, whereas in the second case, the transaction is aborted
- 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.

Types of failures

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

    Also occur because of erroneous parameter values
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 may decide to abort a transaction because itviolates serializability or several transactions are in a state of deadlock

5. Disk failure:

- Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.

6. Physical problems and catastrophes:

- refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake

- Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6.
- Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure.
- Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

5.3 Transaction and System Concepts

5.3.1 Transaction States and Additional Operations

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system keeps track of start of a transaction, termination, commit or aborts.
  - **BEGIN_TRANSACTION**: marks the beginning of transaction execution
  - **READ or WRITE**: specify read or write operations on the database items that are executed as part of a transaction
  - **END_TRANSACTION:** specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution
  - **COMMIT_TRANSACTION**: signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone
  - **ROLLBACK**: signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be **undone**

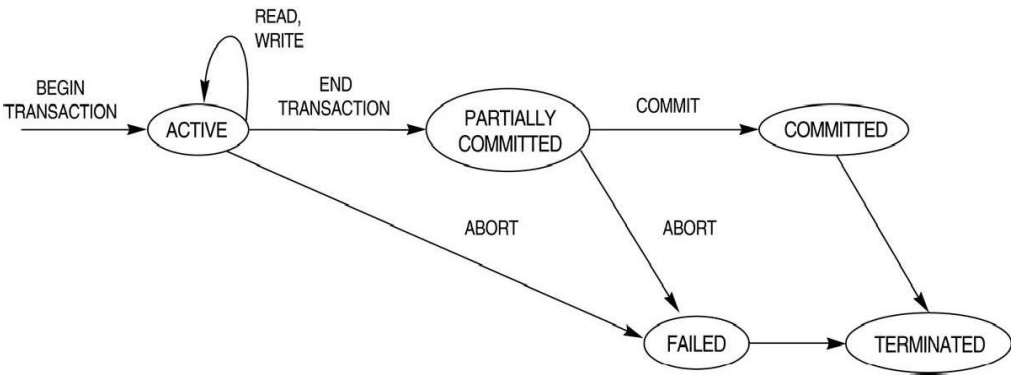


Figure: State transition diagram illustrating the states for transaction execution

- A transaction goes into **active state** immediately after it starts execution and can execute read and write operations.

- When the transaction ends it moves to **partially committed** state.
- At this end additional checks are done to see if the transaction can be committed or not. If these checks are successful the transaction is said to have reached commit point and enters **committed state.** All the changes are recorded permanently in the db.
- 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 operation.
- Terminated state corresponds to the transaction leaving the system. All the information about the transaction is removed from system tables.

•

5.3.2 The System Log

- **Log or Journal** keeps track of all transaction operations that affect the values of database items

•This information may be needed to permit recovery from transaction failures.

The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure

•one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory 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*.

- In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures
- The following are the types of entries—called **log records—**that are written to the log file and the corresponding action for each log record.
- In these entries, T refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:
  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.

5.3.3 Commit Point of a Transaction:

- **Definition a Commit Point:**
  - A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
  - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
  - The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:**
  - Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

5.3.4 DBMS specific buffer Replacement policies

Domain Separation(DS) method

• DBMS cache is divided into separate domains, each handles one type of disk pages and replacements within each domain are handled via basic LRU page replacement.

• LRU is a **static** algorithm and does not adopts to dynamically changing loads because the number of available buffers for each domain is predetermined.

• **Group LRU** adds dynamically load balancing feature since it gives each domain a priority and selects pages from lower priority level domain first for replacement.

- **Hot Set Method:**

  This is useful in queries that have to scan a set of pages repeatedly.

  - The hot set method determines for each db processing algorithm the set of disk pages that will be accessed repeatedly and it does not replace them until their processing is completed.

The DBMIN method:

- uses a model known as QLSM (Query Locality set model), which predetermines the pattern of page references for each algorithm for a particular db 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.

5.4 Desirable Properties of Transactions

- Transactions should possess several properties, often called the **ACID** properties

A **Atomicity:** a transaction is an atomic unit of processing and it is either performed entirely or not at all.

C **Consistency Preservation:** a transaction should be consistency preserving that is it must take the database from one consistent state to another.

I **Isolation/Independence:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executed concurrently.

D **Durability (or Permanency):** if a transaction changes the database and is committed, the changes must never be lost because of any failure.

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 generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.

The **isolation** property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks

Durability is the responsibility of recovery subsystem.

5.5 Characterizing Schedules Based on Recoverability

- **schedule** (or **history):** the order of execution of operations from all the various transactions
- **Schedules (Histories) of Transactions:** A schedule S of n transactions $T_1, T_2 \ldots\ldots T_n$ is a sequential ordering of the operations of the n transactions.
  - The transactions are interleaved
- 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*; and

  (3) *at least one* of the operations is a write_item(*X*)
- **Conflicting operations:**
  - $r_1(X)$ conflicts with $w_2(X)$ } Read write conflict
  - $r_2(X)$ conflicts with $w_1(X)$ }
  - $w_1(X)$ conflicts with $w_2(X)$ Write conflict
  - $r_1(X)$ do not conflicts with $r_2(X)$

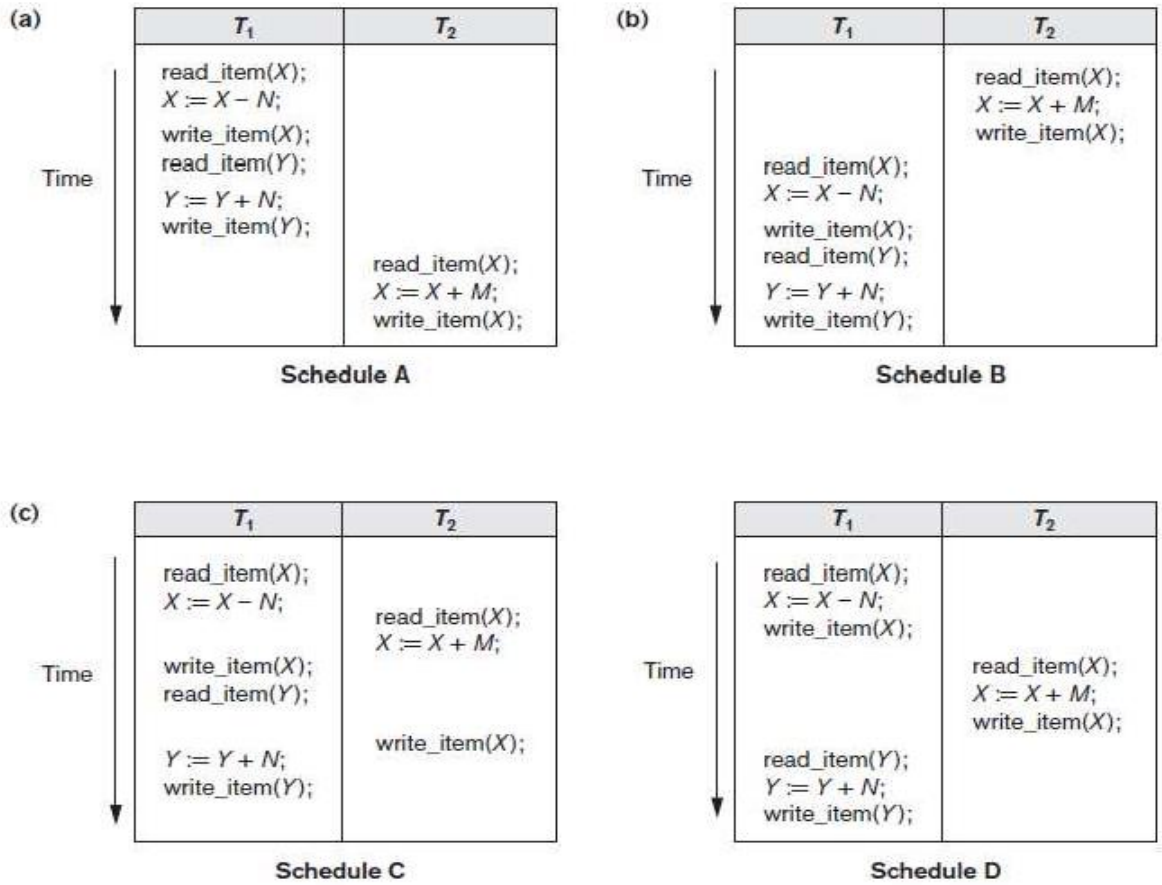
Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
  - Example:
    - S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;
    - S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2;
- **Cascadeless schedule**:
  - One where every transaction reads only the items that are written by committed transactions.
- **Schedules requiring cascaded rollback**:
  - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Strict Schedules**:
  - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

5.6 Characterizing Schedules Based on Serializability

- schedules that are always considered to be correct when concurrent transactions are executing are known as **serializable** schedules
- Suppose that two users for example, two airline reservations agents submit to the DBMS transactions T1 and T2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:
  1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).
  2. Execute all the operations of transaction *T*2 (in sequence) followed by all the operations of transaction *T*1 (in sequence).

Figure 21.5
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.



- **Serial schedule:**
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
    - Otherwise, the schedule is called nonserial schedule.
- **Serializable schedule:**
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
- **Result equivalent:**
  - Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent:**
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable:**
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.
- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

5.6.1 Testing conflict serializability of a Schedule S

For each transaction Ti participating in schedule S,create a node labeled Ti in the precedence graph.

For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create an edge (Ti→ Tj) in the precedence graph.

For each case in S where Tj executes a write_item(X) after Ti executes a read_item (X) ,create an edge (Ti→ Tj) in the precedence graph.

For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create an edge (Ti→ Tj) in the precedence graph.

The schedule S is serializable if and only if the precedence graph has no cycles.

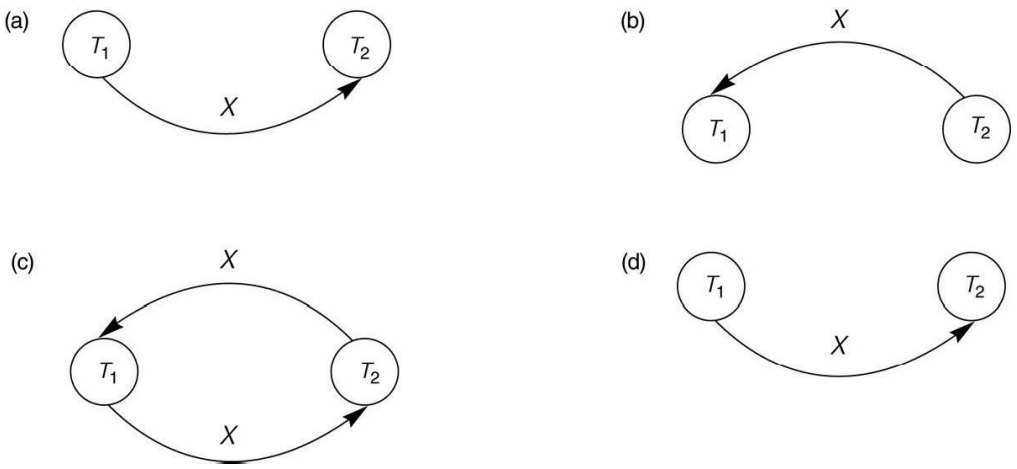


Fig: Constructing the precedence graphs for schedules *A* and *D* from fig 21.5 to test for conflict serializability.

(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*).

- Another example of serializability testing. (a) The READ and WRITE operations of three transactions T_1, T_2, and T_3.

(a)

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

(b)

transaction T_1	transaction ‹2	transaction T_3
	read item (7); read_item (/; write item *(Y),'*	
		read item (V); read_item (Z);
read_item *(X),* write item ( );		
		write item (7), write_item (Z);
	read item (J);	
read_item *(Y);* write_item (;		
	write_item ( );	

Time

Schedule E

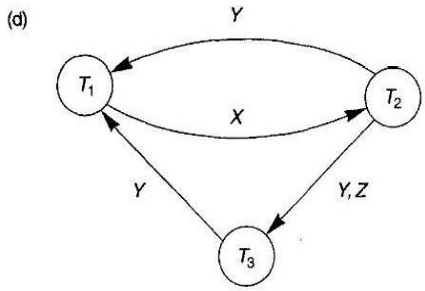
(c)

transaction f/	transaction 2	transaction *Ty*
		read_item *(Y),* read item (Z);
read_item (z); wfi1e_item (X);		
		write item (7); write_item (7);
	read item (2);	
read_item *(Yj;* write_item (;		
	read item *(Y),* write_item (*Y),* read_item ( ); write item *(Xj;*	

Time

Schedule F

- Precedence graph for schedule E



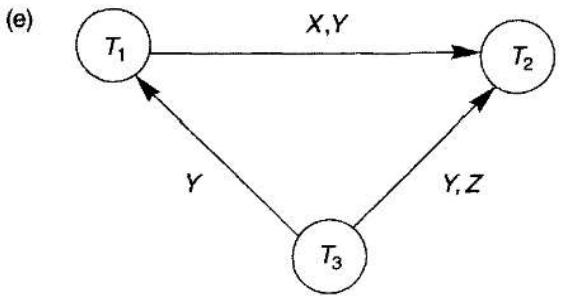
Equivalent serial schedules

None

Reason

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

- Precedence graph for schedule F



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

5.7 Transaction Support in SQL

- The basic definition of an SQL transaction is, it is a logical unit of work and is guaranteed to be atomic
- 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
- With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered
- Every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK
- Every transaction has certain characteristics attributed to it and are specified by a SET TRANSACTION statement in SQL

- The characteristics are :
  - **The access mode**
    - can be specified as READ ONLY or READ WRITE
    - The default is READ WRITE
    - A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed
    - A mode of READ ONLY, as the name implies, is simply for data retrieval.
  - **The diagnostic area size**
    - 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**
  - specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE
  - The default isolation level is SERIALIZABLE
  - 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:
    1. **Dirty read.** A transaction *T*1 may read the update of a transaction *T*2, which has not yet committed. If *T*2 fails and is aborted, then *T*1 would have read a value that does not exist and is incorrect.
    2. **Nonrepeatable read.** A transaction *T*1 may read a given value from a table. If another transaction *T*2 later updates that value and *T*1 reads that value again, *T*1 will see a different value.
    3. **Phantoms.** A transaction *T*1 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 *T*2 inserts a new row that also satisfies the WHERE-clause condition used in *T*1, into the table used by *T*1. If *T*1 is repeated, then *T*1 will see a phantom, a row that previously did not exist.

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

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

```
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: ... ;
```

- The transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2
- If an error occurs on any of the SQL statements, the entire transaction is rolled back
- This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.