Lecture Notes

DATABASE MANAGEMENT SYSTEM (BCS403)

Module – 4

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.

Textbook 1: Ch 7.1 to 7.3, Ch 20.1 to 20.6 RBT: L1, L2, L3

Teaching-Learning Process	Chalk& board, Problem based learning

More SQL: Complex Queries, Triggers, Views, and Schema Modification

1. More Complex SQL Retrieval Queries

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 *unknown* (value exists but is not known, or it is not known whether or not the value exists),

Value *not available* (value exists but is purposely withheld),

Value *not applicable* (the attribute does not apply to this tuple or is undefined for this tuple).

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database. An example of the other case of unknown would be NULL for a person's home phone because it is not known whether or not the person has a home phone.
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 because it does not apply to that person.

Table 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		

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

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

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 another SQL query. That other query is called the **outer query**.

Query: Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

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');

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

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

In addition to the IN operator, a number of other comparison 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*.

Query: 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);

The SQL statement 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**.

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

```
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 );
```

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**.

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

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

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.

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

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);

EXISTS and NOT EXISTS are typically used in conjunction with a *correlated* nested query.

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

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

Query: 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);

The query: 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));

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

SELECT Lname, Fname **FROM** EMPLOYEE
WHERE NOT EXISTS (**SELECT** * **FROM** WORKS_ON B
WHERE (B.Pno IN (**SELECT** Pnumber **FROM** PROJECT
WHERE Dnum = 5)
AND
NOT EXISTS (**SELECT** * **FROM** WORKS_ON C
WHERE C.Essn = Ssn
AND C.Pno = B.Pno)));

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.

Query: 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);

Query: For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate 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;

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.

Query: Retrieve the name and address of every employee who works for the 'Research' department.

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';

The default type of join in a joined table is called an **inner join.**

If the user requires that all employees be included, a different type of join called **OUTER JOIN** must be used explicitly.

Query: For each employee, retrieve the employee's first and last name and the

first and last name of his or her immediate supervisor.

SELECT E.Lname **AS** Employee_name, S.Lname **AS** Supervisor_name
FROM (EMPLOYEE **AS** E **LEFT OUTER JOIN** EMPLOYEE **AS** S
ON E.Super_ssn = S.Ssn);

In some systems, a different syntax was used to specify outer joins by using the comparison operators + =, = +, and + = + for left, right, and full outer join, respectively, when specifying the join condition.

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

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**.

Query: 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';

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.

A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.

Query: 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;

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

Query: 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';

Queries: Retrieve the total number of employees in the company and the number of employees in the 'Research' department.

Q1: SELECT COUNT (*) FROM EMPLOYEE;
Q2: SELECT COUNT (*) **FROM** EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER **AND** DNAME = 'Research';

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

Q3: SELECT COUNT (**DISTINCT** Salary) **FROM** EMPLOYEE;

Query: To retrieve the names of all employees who have two or more **dependents, we can write the following**:

Q4: SELECT Lname, Fname **FROM** EMPLOYEE **WHERE** (**SELECT COUNT** (*)
FROM DEPENDENT **WHERE** Ssn = Essn) > = 2;

Grouping: The GROUP BY and HAVING Clauses

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).

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

Q1: SELECT Dno, **COUNT** (*), **AVG** (Salary) **FROM** EMPLOYEE
GROUP BY Dno;

Query2: For each project, retrieve the project number, the project name, and the number of employees who work on that project.

Q2: SELECT Pnumber, Pname, **COUNT** (*) **FROM** PROJECT, WORKS_ON
WHERE Pnumber = Pno **GROUP BY** Pnumber, Pname;

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.

Query: 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.

Q3: SELECT Pnumber, Pname, **COUNT** (*) **FROM** PROJECT, WORKS_ON
WHERE Pnumber = Pno **GROUP BY** Pnumber, Pname
HAVING COUNT (*) > 2;

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

Q4: SELECT Pnumber, Pname, **COUNT** (*) **FROM** PROJECT, WORKS_ON, EMPLOYEE **WHERE**
Pnumber = Pno **AND** Ssn = Essn **AND** Dno = 5
GROUP BY Pnumber, Pname;

Query: 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.

Q5: SELECT Dno, **COUNT** (*) **FROM** EMPLOYEE **WHERE** Salary>40000 **AND** Dno **IN** (**SELECT** Dno **FROM** EMPLOYEE **GROUP BY** Dno **HAVING COUNT** (*) > 5) **GROUP BY** Dno;

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; it is somewhat similar to creating a view that will be used only in one query and then dropped.

Q: WITH BIGDEPTS (Dno) **AS** (**SELECT** Dno **FROM** EMPLOYEE
GROUP BY Dno **HAVING COUNT** (*) > 5) **SELECT** Dno, **COUNT** (*) **FROM** EMPLOYEE **WHERE** Salary>40000 **AND** Dno **IN** BIGDEPTS **GROUP BY** Dno;

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.

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 ;

This syntax was added in SQL:99 to allow users the capability to specify a recursive query in a declarative manner. An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor.

WITH RECURSIVE SUP_EMP (SupSsn, EmpSsn) **AS**

(**SELECT** SupervisorSsn, Ssn **FROM** EMPLOYEE

UNION

SELECT E.Ssn, S.SupSsn **FROM** EMPLOYEE **AS** E, SUP_EMP **AS** S

WHERE E.SupervisorSsn = S.EmpSsn)

SELECT* **FROM** SUP_EMP;

Discussion and Summary of SQL Queries

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 **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. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

Specifying Constraints as Assertions and Actions as Triggers Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints—those that do not fall into any of the categories — via **declarative assertions**, using the **CREATE ASSERTION** statement. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

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));

Introduction to Triggers in SQL

Another important statement in SQL is CREATE TRIGGER. 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.

R5: 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);

A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three
components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
2. 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. The condition is specified in the WHEN clause of the trigger.

3. 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. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

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 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.

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.

V1: CREATE VIEW WORKS_ON1

AS SELECT Fname, Lname, Pname, Hours

FROM EMPLOYEE, PROJECT, WORKS_ON

WHERE Ssn = Essn **AND** Pno = Pnumber;

V2: 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;

Query: 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 in QV1:

QV1: SELECT Fname, Lname **FROM** WORKS_ON1

WHERE Pname = 'ProductX';

If we do not need a view anymore, we can use the **DROP VIEW** command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement in V1A:

V1A: DROP VIEW WORKS_ON1;

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.

1. **Query modification**
2. **View materialization**

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 QV1 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 view 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 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.

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.

This view update is shown in UV1:

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

Views as Authorization Mechanisms

We describe SQL query authorization statements (GRANT and REVOKE) in detail, when we present database security and authorization mechanisms. Here, we will just give a couple of simple examples to illustrate how views can be used to hide certain attributes or tuples from unauthorized users.

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;

For example, only the first name, last name, and address of an employee may be visible as follows:

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

Schema Change Statements in SQL

The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, types, or constraints.

One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command.

There are two *drop behavior* options: CASCADE and RESTRICT.

DROP SCHEMA COMPANY **CASCADE**; **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.

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.

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.

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

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior.

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.

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

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

One can also change the constraints specified on a table by adding or dropping a named constraint.

ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK **CASCADE**;

ADD CONSTRAINT keyword in the ALTER TABLE statement followed by the new constraint.

Chapter 2:

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.

1.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**.

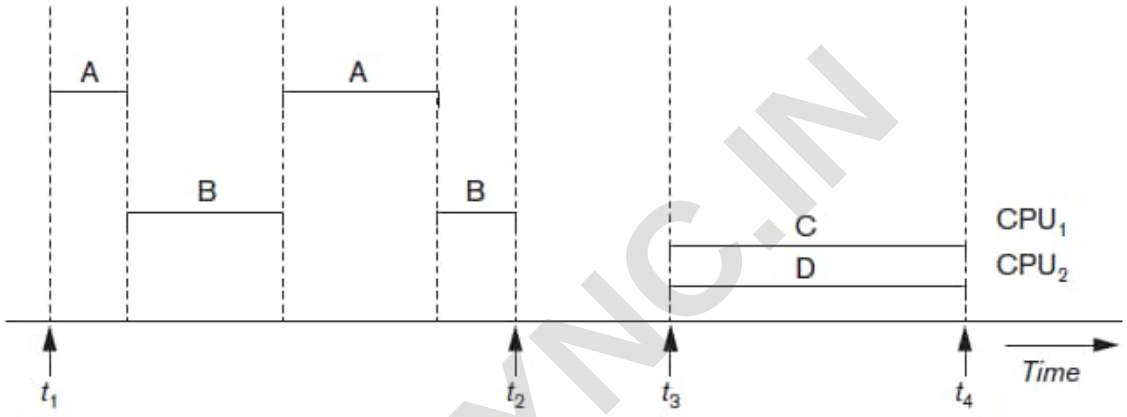
A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems. Example: contact lists, calendars, and document repositories. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. An airline reservations system is used by hundreds of users and travel agents concurrently.

Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming**, which allows the operating system of the computer to execute multiple programs—or **processes**—at the same time.

A single central processing unit (CPU) can only execute at most one process at a time. However, **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**.



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 (update), or retrieval operations.

One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

A single application program may contain more than one transaction if it contains several transaction boundaries.

If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

The *database model* that is used to present transaction processing concepts is simple when compared to the data models, such as the relational model or the object model. 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*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name.

Using this 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.*

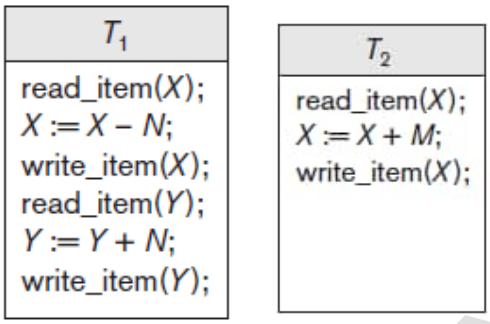
■ **write_item(*X*).** Writes the value of program variable *X* into the database item named *X*.

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*.

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).



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.

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.

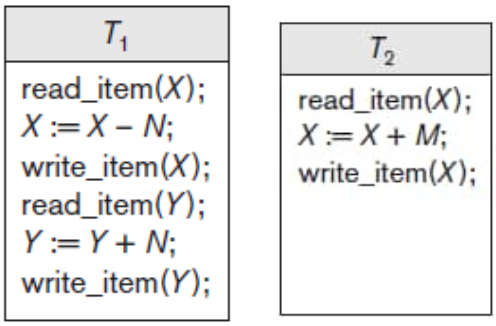
When these buffers are all occupied, and additional database disk blocks must be copied into memory, some **buffer replacement policy** is used to choose which of the current occupied buffers is to be replaced.

Some commonly used buffer replacement policies are **LRU** (least recently used).

Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

Consider a situation in airline reservations database. Some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight. Each record includes the *number of reserved seats* on that flight as a *named (uniquely identifiable) data item*, among other information.



Transaction *T*1, *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 *T*2, just *reserves M* seats on the first flight (*X*) referenced in transaction *T*1.

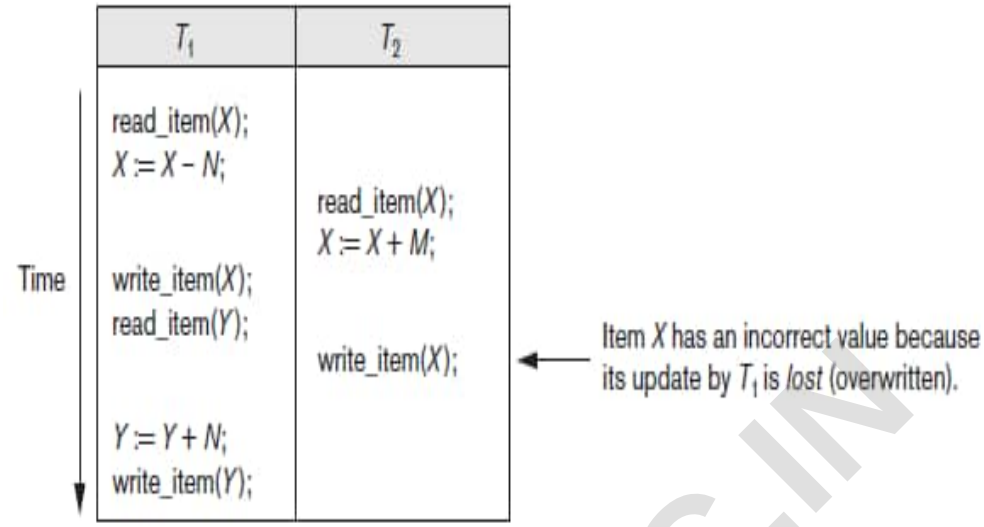
Types of problems

Types of problems may encounter with these two simple transactions if they run concurrently.

1.The Lost Update Problem -

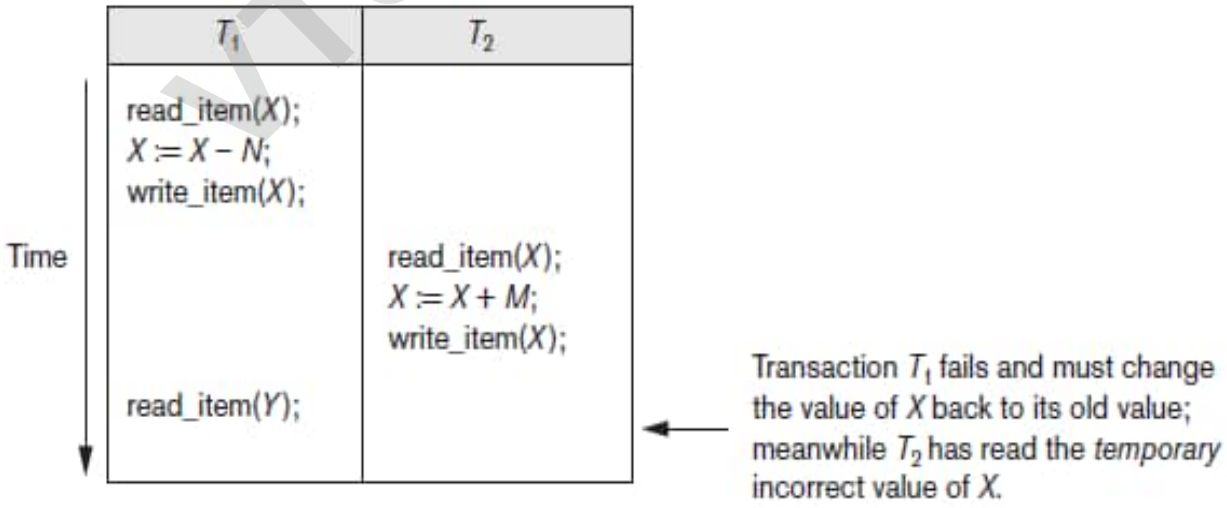
This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Suppose that transactions *T*1 and *T*2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure; then the 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.



2.The Temporary Update (or Dirty Read) Problem.

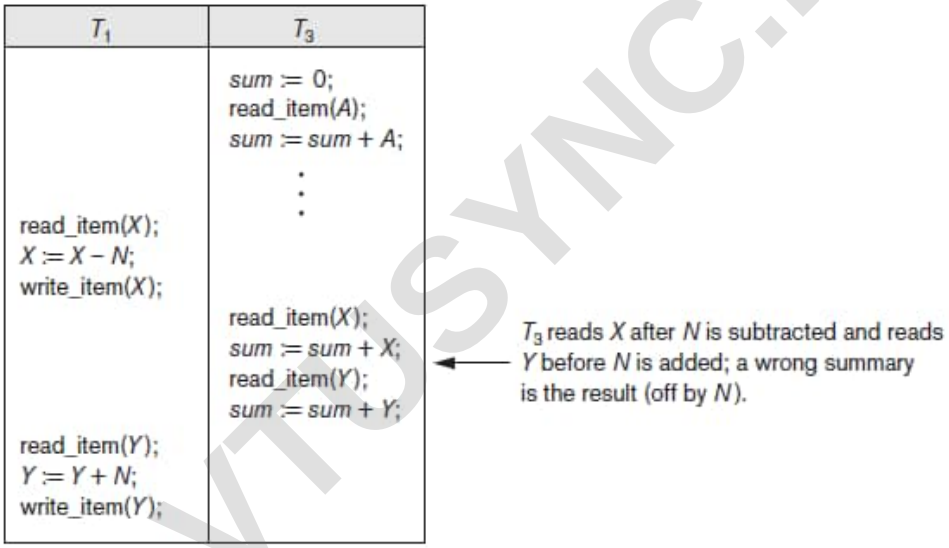
This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value. Figure shows an example where *T*1 updates item *X* and then fails before completion, so the system must roll back *X* to its original value.



3. 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.

For example, suppose that a transaction *T*3 is calculating the total number of reservations on all the flights; meanwhile, transaction *T*1 is executing. If the interleaving of operations shown in Figure occurs, the result of *T*3 will be off by an amount *N* because *T*3 reads the value of *X after N* seats have been subtracted from it but reads the value of *Y before* those *N* seats have been added to it



Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that 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**.

The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of *T* are not, because *the 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.

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. **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.
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, or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

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.

2.Transaction and System Concepts

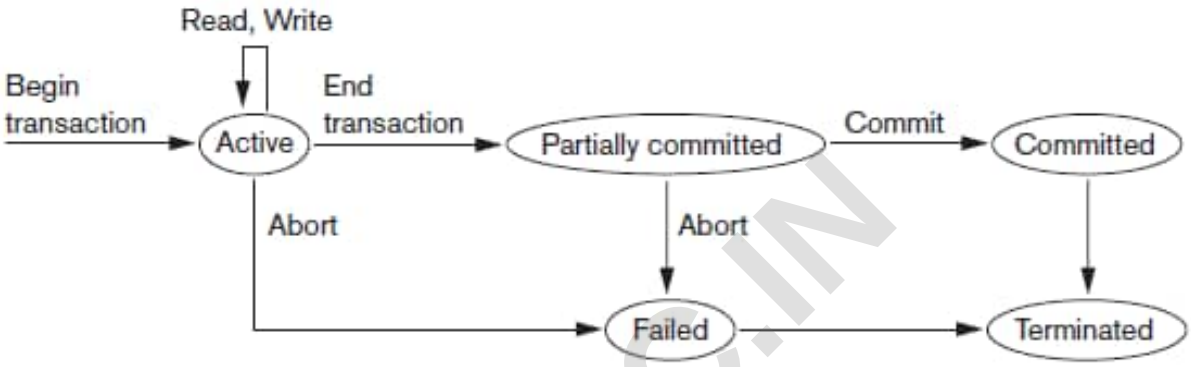
Transaction States and Additional Operations

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 needs to keep track of when each transaction starts, terminates, and commits, or aborts. Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

- BEGIN_TRANSACTION. This marks the beginning of transaction execution.
- READ or WRITE. These specify read or write operations on the database items that are executed as part of a transaction.
- END_TRANSACTION. This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.
- COMMIT_TRANSACTION. This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.

- ROLLBACK (or ABORT). This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be **undone**.



The System Log

To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures.

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

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.

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 system 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.

Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

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 in the DBMS cache are occupied and new disk pages are required to be loaded into main memory from disk, a **page replacement policy** is needed to select the particular buffers to be replaced.

Domain Separation (DS) Method.

In a DBMS, various types of disk pages exist: index pages, data file pages, log file pages, and so on. In this method, the DBMS cache is divided into separate domains (sets of buffers). Each domain handles one type of disk pages, and page replacements within each domain are handled via the basic LRU (least recently used) page replacement.
Although this achieves better performance on average that basic LRU, it is a *static algorithm,* and so 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, which add dynamic load-balancing features.

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.

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.

We discussed various algorithms for relational operations such as SELECT and JOIN. 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 (some queries may reference the same file twice, so there would be a locality set for each file instance needed 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.

The concept of locality set is analogous to the concept of *working set*, which is used in page replacement policies for processes by the operating system but there are multiple locality sets, one for each file instance in the query.

3. Desirable Properties of Transactions

Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

- **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.

- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the 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.

4. 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** (or **history**).

Schedules (Histories) of Transactions

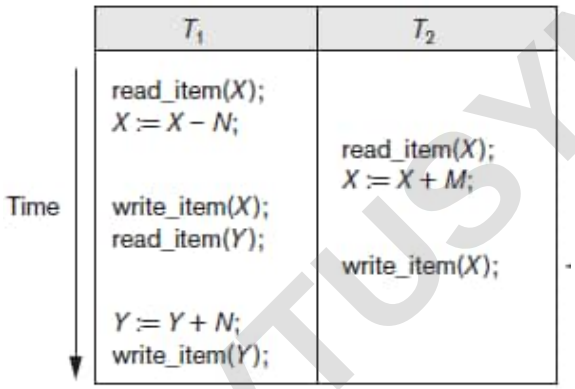
A **schedule** (or **history**) *S* of *n* transactions *T*1, *T*2, ... , *Tn* is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule *S*. However, for each transaction *Ti* that participates in the schedule *S*, the operations of *Ti* in *S* must appear in the same order in which they occur in *Ti*.

The order of operations in *S* is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.

For the purpose of recovery and concurrency control, we are mainly interested in the read_item and write_item operations of the transactions, as well as the commit and abort operations.

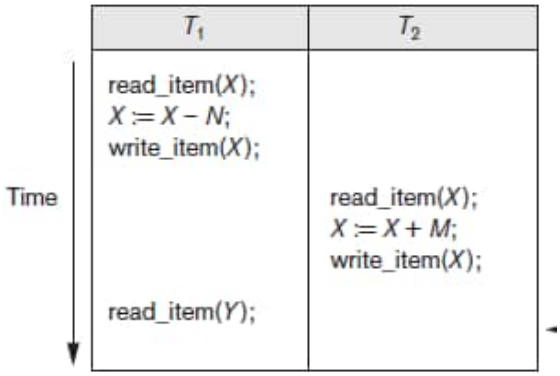
A shorthand notation for describing a schedule uses the symbols *b*, *r*, *w*, *e*, *c*, and *a* for the operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule.

In this notation, the database item *X* that is read or written follows the *r* and *w* operations in parentheses. In some schedules, we will only show the *read* and *write* operations, whereas in other schedules we will show additional operations, such as commit or abort.



The schedule in Figure, which we shall call *Sa*, can be written as follows in this notation:

***Sa*: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*);**



***Sb*: *r*1(*X*); *w*1(*X*); *r*2(*X*); *w*2(*X*); *r*1(*Y*); *a*1;**

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*; and

(3) *at least one* of the operations is a write_item(*X*).

For example, in schedule *Sa*, 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*).

The operaations *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.

Intuitively, two operations are conflicting if changing their order can result in a different outcome.

- **read-write conflict**
- **write-write conflict**

For example, if we change the order of the two operations $r1(X)$; $w2(X)$ to $w2(X)$; $r1(X)$, then the value of X that is read by transaction $T1$ changes, because in the second ordering the value of X is read by $r1(X)$ *after* it is changed by $w2(X)$, whereas in the first ordering the value is read *before* it is changed. This is called a **read-write conflict**.

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 $w1(X)$;$w2(X)$ to $w2(X)$; $w1(X)$.

Some theoretical definitions concerning schedules.

A schedule S of n transactions $T1$, $T2$, … , Tn is said to be a **complete schedule** if the following conditions hold:

1. The operations in S are exactly those operations in $T1$, $T2$, … , Tn, 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 Ti, their relative order of appearance in S is the same as their order of appearance in Ti.
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*.

First, 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. The (partial) schedules Sa and Sb from the preceding section are both recoverable, since they satisfy the above definition.

Consider the schedule Sa′ given below, which is the same as schedule Sa except that two commit operations have been added to Sa:

Sa′: r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;

Sa′ is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory.

consider the two (partial) schedules S_c and S_d that follow:

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

S_d: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;

S_e: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;

Sc 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 *Sc*; 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*.

For the schedule to be recoverable, the *c*2 operation in *Sc* must be postponed until after *T*1 commits, as shown in *Sd*. If *T*1 aborts instead of committing, then *T*2 should also abort as shown in *Se*, because the value of *X* it read is no longer valid.

In *Se*, aborting *T*2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule *Sc*.

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

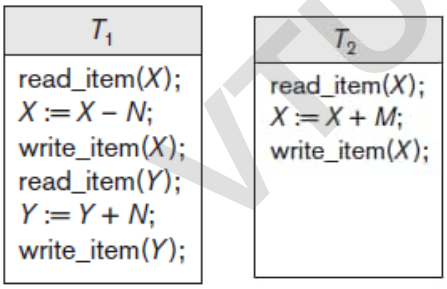
possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

5. Characterizing Schedules Based on Serializability

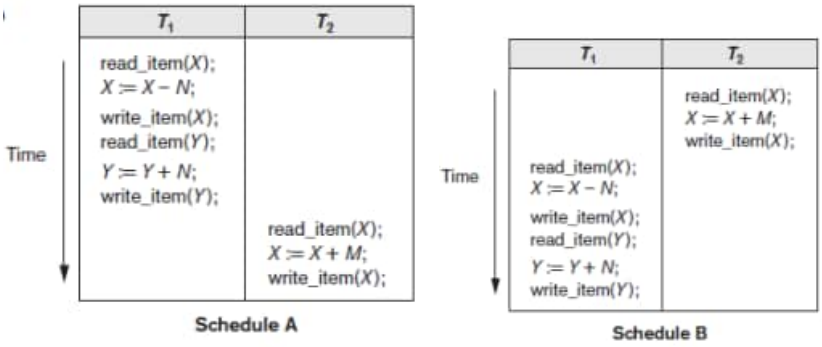
Now we characterize the types of schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions $T1$ and $T2$ in Figure 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 $T2$ (in sequence) followed by all the operations of transaction $T1$ (in sequence).



These two schedules—called *serial schedules*—are shown in Figures, respectively.

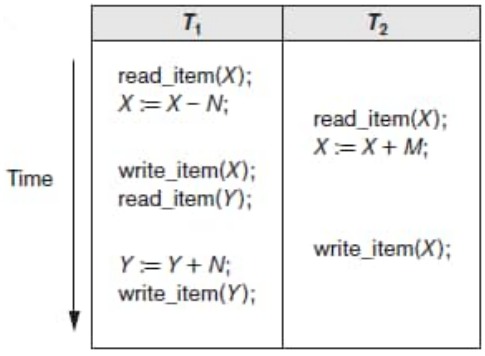


The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

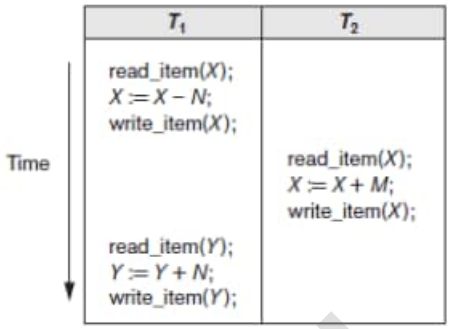
Serial, Nonserial, and Conflict-Serializable Schedules

Formally, 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. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be independent, is that every serial schedule is considered correct.

Schedules A and B in Figures are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.



Schedule C



Schedule D

Schedules C and D in Figure are called *nonserial* because each sequence interleaves operations from the two transactions.

The definition of serializable schedule is as follows: A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions. We will define the concept of equivalence of schedules shortly. Notice that there are n! possible serial schedules of n transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules— those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to any serial schedule and hence are not serializable.

Testing for Serializability of a Schedule

There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not. Most concurrency control methods do not actually test for serializability. Rather protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable. Some methods guarantee serializability in most cases, but do not guarantee it absolutely, in order to reduce the overhead of concurrency control.

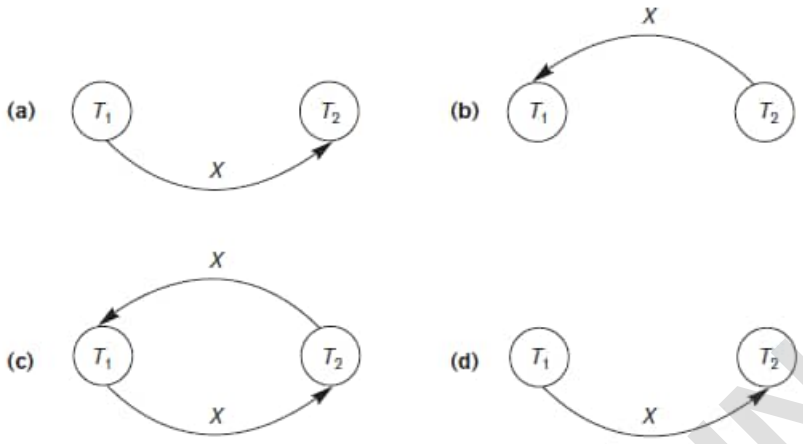
The algorithm looks at only the read_item and write_item operations in a schedule to construct a precedence graph (or serialization graph), which is a directed graph G =

(N, E) that consists of a set of nodes N = {T1, T2, … , Tn } and a set of directed edges E = {e1, e2, … , em }. There is one node in the graph for each transaction Ti in the schedule. Each edge ei in the graph is of the form (Tj → Tk), 1 ≤ j ≤ n, 1 ≤ k ≤ n, where Tj is the starting node of ei and Tk is the ending node of ei. Such an edge from node Tj to node Tk is created by the algorithm if a pair of conflicting operations exist in Tj and Tk and the conflicting operation in Tj appears in the schedule before the conflicting operation in Tk.

Algorithm: Testing Conflict Serializability of a Schedule S

1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
2. 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.
3. 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.
4. 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.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

The algorithm looks at only the read_item and write_item operations in a schedule to construct a precedence graph (or serialization graph), which is a directed graph G = (N, E) that consists of a set of nodes N = {T1, T2, … , Tn } and a set of directed edges E = {e1, e2, … , em }.



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);

Time

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

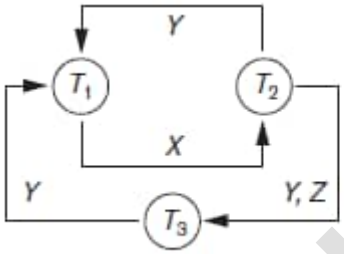
Schedule E

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

read_item(Y);
write_item(Y);
	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	

Time

Schedule F

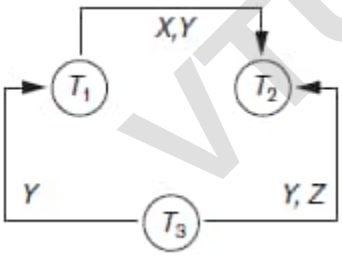


Equivalent serial schedules

None

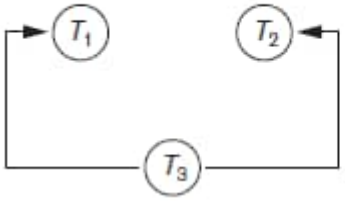
Reason

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



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

6. Transaction Support in SQL

The basic definition of an SQL transaction is similar to our already defined concept of a transaction. That 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. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it.

These characteristics are specified by a SET TRANSACTION statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *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: ... ;
```

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.

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.

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.

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 *r* that also satisfies the WHERE-clause condition used in *T*1, into the table used by *T*1. The record *r* is called a **phantom record** because it was not there when *T*1 starts but is there when *T*1 ends.

Table 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