UNIT- IV

Advanced SQL and Transaction Management

Advanced SQL and Transaction Management

Sessions to be Covered in Unit-IV

- Advanced SQL
- Transaction Management

SESSION-I: TOPICS TO BE COVERED

Advanced SQL

- **Complex Queries, Triggers, Views, and Schema Modification**
  - More Complex SQL Retrieval Queries
  - Specifying Constraints as Assertions and Actions as Triggers
  - Views (Virtual Tables) in SQL
  - Schema Change Statements in SQL.

SESSION-II: TOPICS TO BE COVERED

Transaction Management

- Introduction to transaction processing
- Transaction and system concept
- Desirable properties of transaction
- Transaction support in SQL
- Concurrency control techniques – Two phase Locking techniques for concurrency, timestamp based protocol.

SESSION-I: TOPICS TO BE COVERED

Advanced SQL

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

- More Complex SQL Retrieval Queries
- Specifying Constraints as Assertions and Actions as Triggers
- Views (Virtual Tables) in SQL
- Schema Change Statements in SQL.

More Complex SQL Retrieval Queries

- Additional features allow users to specify more complex retrievals from database:
  - Nested queries, joined tables, outer joins, aggregate functions, and grouping

Comparisons Involving NULL and Three-Valued Logic

- SQL has various rules for dealing with NULL values.

- Meanings of NULL
  - **Unknown value**
  - **Unavailable or withheld value**
  - **Not applicable attribute**

- It is often not possible to determine which of the meanings is intended for NULL
  - Ex: A NULL for the home phone of a person can have any of the three meanings
  - Hence, SQL does not distinguish between the different meanings of NULL.
  - Each individual NULL value considered to be different from every other NULL value

- 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.

- So, SQL uses a three-valued logic:
  - TRUE, FALSE, and UNKNOWN

Table 5.1 shows the resulting TRUTH values.

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

- SQL allows queries that **check whether** an attribute value is NULL
  - IS or IS NOT NULL

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

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

Nested Queries, Tuples, and Set/Multiset Comparisons

- **Nested queries**
  - Some queries require that existing values in the database to 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
- **Comparison operator IN**
  - Compares value *v* with a set (or multiset) of values *V*
  - Evaluates to TRUE if *v* is one of the elements in *V*

Example: The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, while 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, uses the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query

```
Q4A:  SELECT   DISTINCT Pnumber
      FROM     PROJECT
      WHERE    Pnumber IN
               ( SELECT     Pnumber
                 FROM       PROJECT, DEPARTMENT, EMPLOYEE
                 WHERE      Dnum=Dnumber AND
                            Mgr_ssn=Ssn AND Lname='Smith' )
               OR
               Pnumber IN
               ( SELECT     Pno
                 FROM       WORKS_ON, EMPLOYEE
                 WHERE      Essn=Ssn AND Lname='Smith' );
```

- 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** Operator.
- The two keywords ANY and SOME have the same effect.
- Other operators that can be combined with ANY (or SOME) which include >, >=, <=, and <> (Not equal to).

- 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.

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

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

- Avoid **potential errors and ambiguities**
  - Create tuple variables (aliases) for all tables referenced in SQL query

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

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

- **Correlated** nested query : 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.
  - Evaluated once for each tuple in the outer query
  - One time comparison evaluation enough.

- 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.

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

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

The EXISTS and UNIQUE Functions in SQL

- EXISTS function
  - Check whether the **result of a correlated nested query** is empty or not

- EXISTS and NOT EXISTS
  - Typically used in conjunction with a correlated nested query

- SQL function UNIQUE(Q)
  - Returns TRUE if there are **no duplicate tuples in the result of query Q**

```
Q16B: 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);
```

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

Explicit Sets and Renaming of Attributes in SQL

- Can use **explicit set of value**s in WHERE clause
- Use qualifier **AS** followed by desired new name
  - Rename any attribute that appears in the result of a query

```
Q8A:    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

- **Joined table**
  - Permits users **to specify a new table that resulting from a join operation** in the FROM clause of a query
- The FROM clause in Q1A
  - Contains a single joined table

```
Q1A:    SELECT    Fname, Lname, Address
        FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE     Dname='Research';
```

- Specify different **types of join**
  - NATURAL JOIN
  - Various types of OUTER JOIN

- **NATURAL JOIN** on two relations R and S
  - No join condition specified
  - Implicit **EQUIJOIN** condition for **each pair of attributes with same name** from R and S

- **Inner join**
  - Default type of join in a joined table
  - Tuple is included in the result only if a **matching tuple exists in the other relation**

- **Left Outer Join**
  - Every tuple in left table(A) must appear in result
  - If no matching tuple
    - Padded (Filled out) with NULL values for attributes of right table

- Ex – Table A = 10 Records B =5

EMP-ID	EMP- NAME	EMP-ID	SALARY
1	A	1	A
2	B	2	B
.		5	E
.		11	H
10	J	12	I

- **Right Outer Join**
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for the attributes of left table
- **Full Outer Join**
  - Can nest join specifications.
  - Both table records retrieved. But unmatched records retrieved with NULL

Aggregate Functions in SQL

- Used to **summarize information from multiple tuples into a single-tuple** as summary
- **Grouping**
  - Create subgroups of tuples before summarizing
- **Built-in aggregate functions**
  - **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**
- Functions can be used in the **SELECT clause** or in a **HAVING clause**

- NULL values discarded when aggregate functions are applied to a particular co

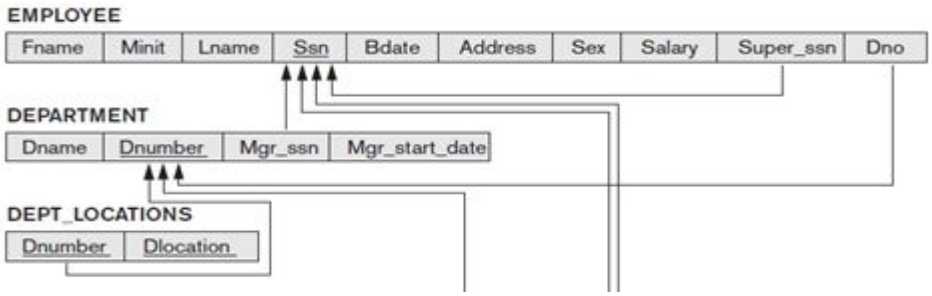
Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

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

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

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

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



Grouping: The GROUP BY and HAVING Clauses

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently as per requirement

- **GROUP BY** clause
  - Specifies grouping attributes
- If NULLs exist in grouping attribute
  - Separate group which created for all tuples with a NULL value in grouping attribute

- **HAVING** clause
  - Provides **a condition on the summary** information

Query 28. 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.

```
Q28:  SELECT   Dnumber, COUNT (*)
      FROM     DEPARTMENT, EMPLOYEE
      WHERE    Dnumber=Dno AND Salary>40000 AND
               ( SELECT       Dno
                 FROM         EMPLOYEE
                 GROUP BY Dno
                 HAVING       COUNT (*) > 5)
```

Discussion and Summary of SQL Queries

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

Specifying Constraints as Assertions and Actions as Triggers

- **CREATE ASSERTION**
  - Specify **additional types of constraints** outside scope of **built-in relational model constraints.**
    - (such as domain constraints, key constraints, and referential Integrity constraints.)

- **CREATE TRIGGER**
  - Specify **automatic actions** that database system will perform when certain events and conditions occur

(A **trigger** is a special type of **stored procedure** that automatically runs when an event occurs in the database server. DML **triggers** run when a user tries to modify data through a data manipulation language (DML) event.

DML events are INSERT, UPDATE, or DELETE statements on a table or view)

- **CREATE ASSERTION**
  - Used to specify a query that **selects any tuples that violate the desired condition**
  - Use only in cases where it is not possible to **use CHECK on attributes** and domains
  - 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

- **CREATE TRIGGER statement**
  - Used to monitor the database
- Typical trigger has three components :
  - **Event(s)-**These are usually database update operations that are explicitly applied to the database**,**
  - **Condition -** The condition that determines whether the rule action should be executed
  - **Action -** 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.

- **Example**

```
CREATE TRIGGER SALARY-VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR-SSN      EVENTS ( BEFORE
Alternate     ON EMPLOYEE                                  for AFTER )
FOR EACH ROW
    WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE    CONDITION
          SSN = NEW.SUPERVISOR-SSN ) )
INFORM-SUPERVISOR(NEW.Supervisor-ssn,
                  NEW.Ssn );                              ACTION
```

Views (Virtual Tables) in SQL

- **Concept of a view in SQL**
  - Single table derived from other tables
  - Considered to be a virtual table

- **Specification of Views in SQL**

  - **CREATE VIEW** command
    - Give table name, list of attribute names, and a query to specify the contents of the view

Example (Virtual)

```
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;
```

Figure 5.2
Two views specified on the database schema of Figure 3.5.

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

- Used to specify SQL queries on a view
- View always up-to-date.
  - It is responsibility of the DBMS and not the user
- **DROP VIEW** command
  - Dispose of a view

View Implementation, View Update, and Inline Views

- It is a **Complex problem of efficiently implementing a view for querying**

  (Implementation of a view for querying complex problems)
- Also **Query modification** approach
  - Modify view query into a query on **underlying base tables**

- **Disadvantage**: inefficient for views defined via complex queries that are time-consuming to execute

(views are inefficiently defined through complex queries and that are time consuming)

View Implementation

- **View materialization approach**
  - Physically create a **temporary view table** when the **view is first queried**
  - Keeps that table **on the assumption** that other queries on the view will follow
  - Requires **efficient strategy** for automatically updating the view table when the base tables are updated

- **Incremental update strategies**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table

Schema Change Statements in SQL.

- **Schema evolution commands**
  - Can be done while the database is operational
  - Does not require recompilation of the database schema

- **The DROP Command**

  - DROP command
    - Used to **drop named schema elements**, such as tables, domains, or constraint
  - Drop behavior options:
    - CASCADE and RESTRICT
  - Example:
    - DROP SCHEMA COMPANY CASCADE;

The ALTER Command

- **Alter table actions** include:
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints
- Example:
  - ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);
- To drop a column
  - Choose either CASCADE or RESTRICT

- Change **constraints specified on a table**
  - Add or drop a named constraint

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

- **Summary**
  - Complex SQL:
    - Nested queries, joined tables, outer joins, aggregate functions, grouping
  - CREATE ASSERTION and CREATE TRIGGER
    - Views
  - Virtual or derived tables

SESSION-II: TOPICS TO BE COVERED

Transaction Management

- Introduction to transaction processing
- Transaction and system concept
- Desirable properties of transaction
- Transaction support in SQL
- Concurrency control techniques – Two phase Locking techniques for concurrency, timestamp based protocol.

Introduction to transaction processing

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser System**:
  - Many users can access the system concurrently.
- **Concurrency**
  - **Interleaved processing**:
    - Concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**:
    - Processes are concurrently executed in multiple CPUs.

Example

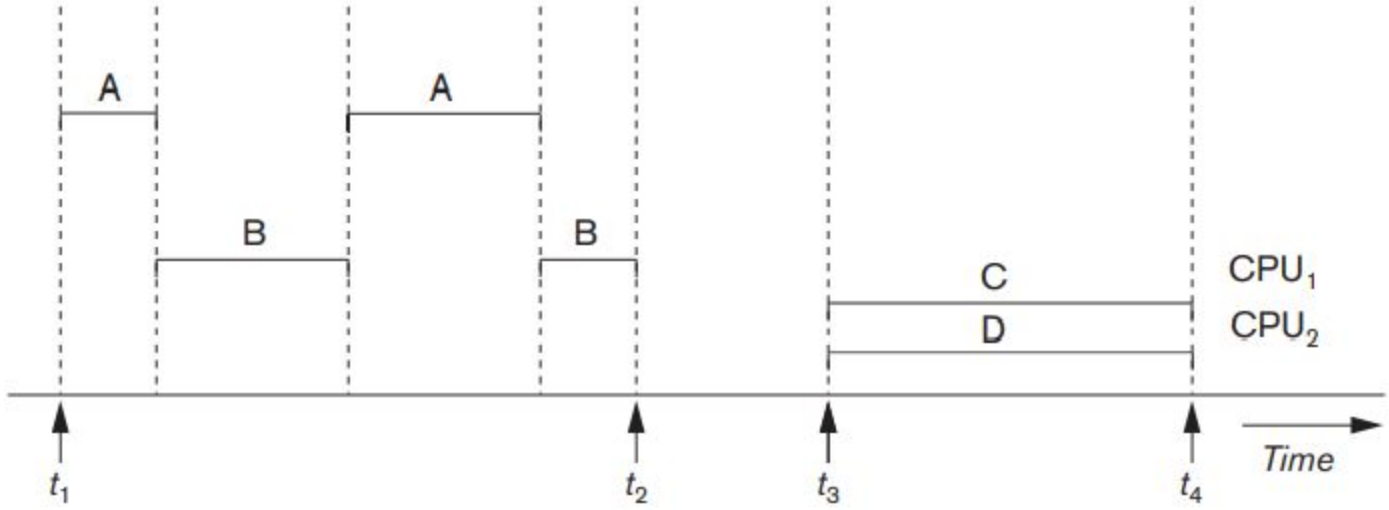


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

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

Example: A SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database** is a collection of named data items
- **Granularity** of data - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
  - **read-item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
  - **write-item(X)**: Writes the value of program variable X into the database item named X.

Note: **Granularity** is the **level of detail** at which data are stored in a database. When the same data are represented in multiple databases, the granularity may differ.

READ AND WRITE OPERATIONS:

- **Basic unit of data transfer** from the disk to the computer main memory is one block.

- In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

READ OPERATION

- read-item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the **buffer to the program variable named X**.

WRITE OPERATION

- **write-item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the **program variable named X into its correct location** in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Two sample transactions

- Here two sample transactions:
  - (a) Transaction T1
  - (b) Transaction T2

(a) T_1

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

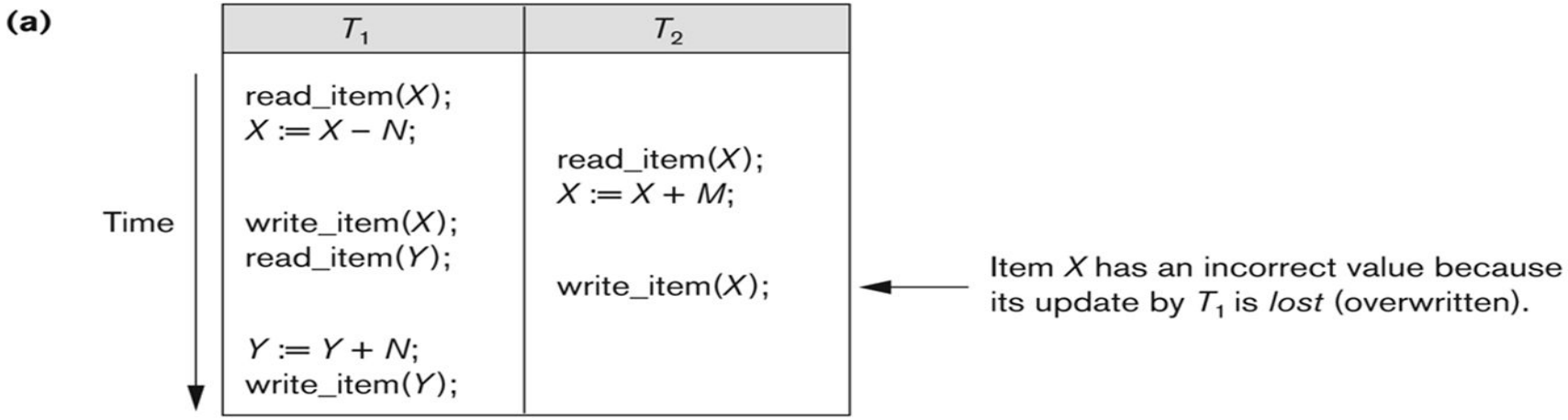
(b) T_2

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

Why Concurrency Control is needed:

- **The Lost Update Problem**
  - This occurs **when two transactions that access the same database items have their operations interleaved in a way** that makes the value of some database item incorrect.

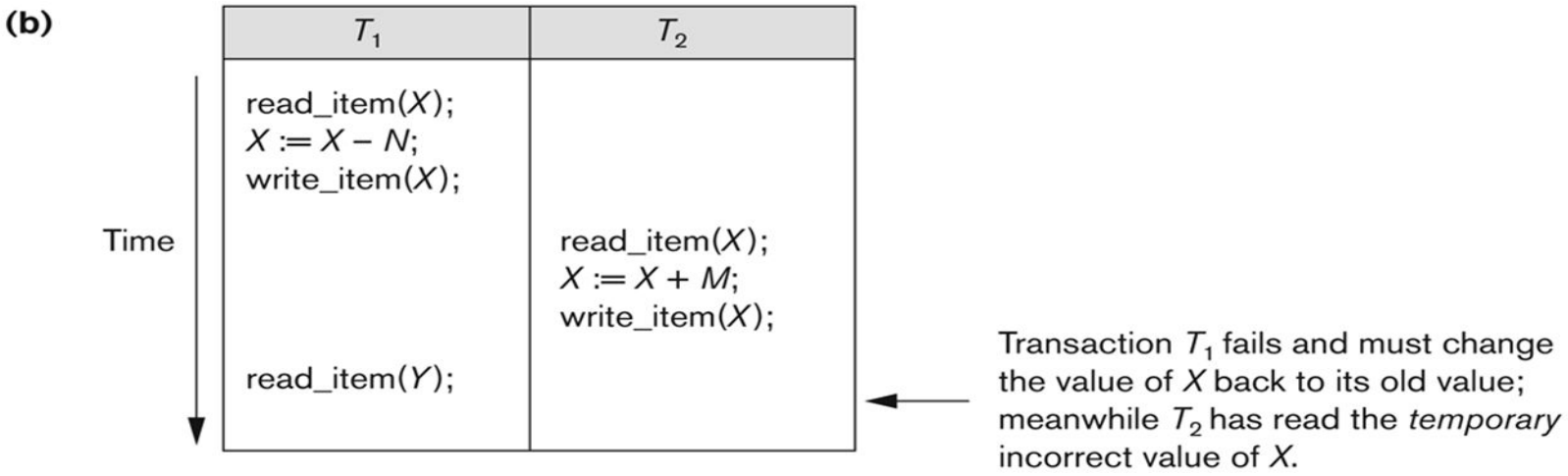
Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



- For example, if X = 80 at the start and N = 5, and M = 4
- The final result should be X = 79.
- However, in the interleaving of operations shown in Figure in previous slide
- it is X = 84 because the update in T1 that removed the five seats from X was lost.

- **The Temporary Update (or Dirty Read) Problem**
  - This occurs **when one transaction updates a database item and then the transaction fails** for some reason.
  - The **updated item is accessed by another transaction** before it is changed back to its original value.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



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

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	. . .

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

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

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

Why **recovery** is needed:

(What causes a Transaction to fail)

1. A computer failure (system crash):

   A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the **computer's internal** memory may be lost.

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**. In addition, the **user may interrupt the transaction** during its execution.

3. Local errors or exception conditions detected by the transaction:

Certain conditions force **cancellation of the transaction** due to data for the transaction may not be found.

For example: A condition, such as **insufficient account balance in a banking database**, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

4. Concurrency control enforcement:

The concurrency control method may decide to **abort the transaction** and to be restarted later, because it violates serializability or because **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**. 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**, **overwriting disks or tapes by mistake** etc.

Transaction and system concepts

- A **transaction** is an **atomic unit of work** that is either completed in its entirety or not done at all (partially completed)
  - For recovery purposes, the system needs **to keep track of when the transaction starts, terminates, and commits or aborts.**
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

State transition diagram illustrating the states for transaction execution

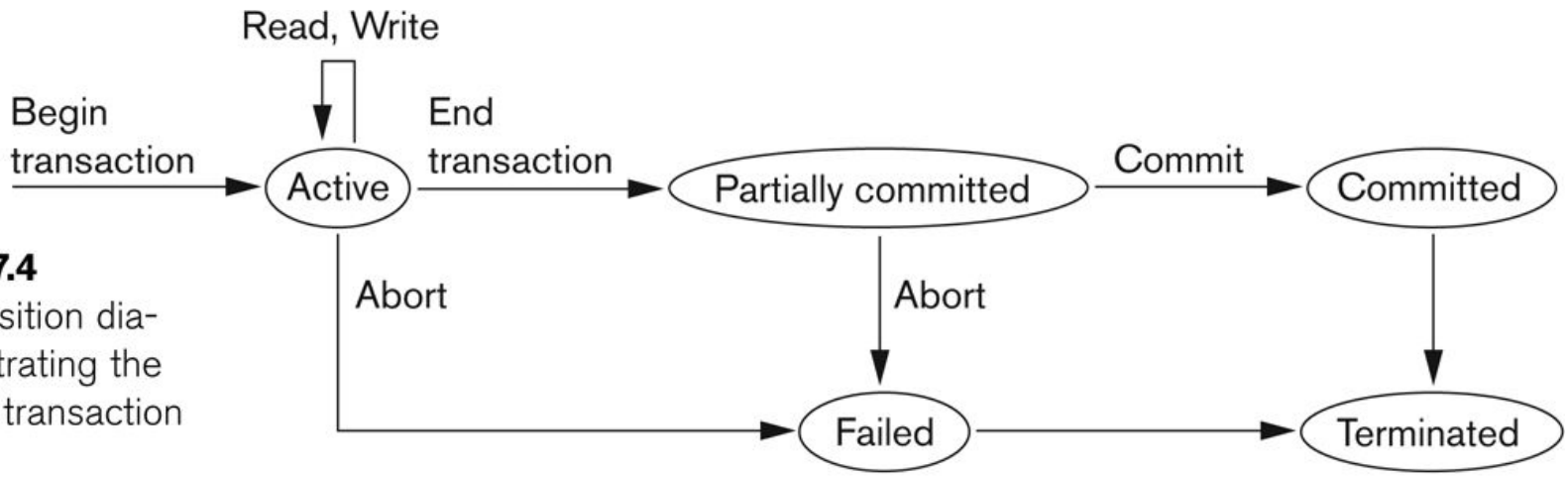


Figure 17.4
State transition diagram illustrating the states for transaction execution.

- **Recovery manager keeps track of the following operations:**
  - **begin-transaction**: This marks the beginning of transaction execution.
  - **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.
  - **end-transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it may be,
      - Necessary to check **whether the changes introduced by the transaction can be permanently applied to the database**

        (or)

      - **whether the transaction has to be aborted** because it violates concurrency control or for some other reason.

- **commit-transaction**: This will signals (indicates) - **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**

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a **single operation** rather than to a **whole transaction**.
  - **redo**: This specifies that certain **transaction operations must be redone** to ensure that all the operations of a committed transaction have been applied successfully to the database

- **The System Log**
  - **Log** (record) : The log **keeps track of all transaction operations** that **affect the values of database items**. Because,
    - This information may be needed to permit recovery from transaction failures.
    - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
    - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

- If T in the following discussion refers to a **unique transaction-id** that is generated automatically by the system and is used to identify each transaction:
- **Types of log record:**
  - [start-transaction,T]: It records that transaction T has started execution.
  - [write-item,T,X,old-value,new-value]: It records that transaction T has changed the value of database item X from old-value to new-value.
  - [read-item,T,X]: It records that transaction T has read the value of database item X.
  - [commit,T]: It records that transaction T has completed successfully, and confirms that its effect can be committed (recorded permanently)

- Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log*, whereas other protocols require these entries for recovery.
- Strict protocols require simpler write entries that do not include new-value

Recovery using log records:

- If the **system crashes**, we can recover to a **consistent database state** by examining the log and using one of the techniques as follows.
  1. Because the **log contains a record of every write operation that changes the value of some database item**, it is possible to **undo** the effect of these write operations of a transaction T by **tracing backward** through the log and resetting all items changed by a write operation of T to their old-values.
  2. We can also **redo** the effect of the write operations of a transaction T **by tracing forward** through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new-values.

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. (From commit point to start position)

- **Redoing transactions:**
  - Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (Notice that the log file must be kept on disk)
  - At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.)
- **Force writing a log:**
  - Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk.
  - This process is called force-writing the log file before committing a transaction.

Desirable properties of transaction

ACID properties:

- **Atomicity**: A transaction is an **atomic unit of processing;** it is either performed in its entirety or not performed at all.

- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.

- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary

.

- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost due to its subsequent failure.

Transaction support in SQL

- A **single** SQL statement is always considered to be **atomic**.
  - Either the statement completes execution without error or it fails and leaves the database unchanged.

- With SQL, there is no explicit Begin Transaction statement.
  - Transaction initiation is done implicitly when particular SQL statements are encountered.

- But every transaction must have an explicit end statement, which is either a COMMIT or 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.

- **Access mode**:
  - READ ONLY or READ WRITE.
    - The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.

- **Diagnostic size n,** specifies an integer value n, indicating the number of conditions that can be held simultaneously performed for a transaction in the diagnostic area.

- **Isolation level** <isolation>, where <isolation> can be
- **Non Repeatable Read**
  - READ UNCOMMITTED (Dirty read- Reads Uncommitted values or temporary values),
  - READ COMMITTED (Reads Committed values)

- **REPEATABLE READ or SERIALIZABLE.**
- The default is SERIALIZABLE (May be Read or Write)
  - With SERIALIZABLE: the interleaved execution of transactions will adhere to our concept of serializability.
  - However, if any transaction executes at a lower level, then serializability may be violated

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a transaction which failed.
- **Nonrepeatable Read**:
  - Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
    - Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

- Phantoms:
  - New rows being read using the same read with a condition.
    - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
    - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
    - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

- **Possible violation of serializabilty:**

Isolation level	Type of Violation: Dirty read	nonrepeatable read	phantom
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no

Concurrency control techniques – Two phase Locking techniques for concurrency, timestamp based protocol.

- **Purpose of Concurrency Control**
  - To ensure that the **Isolation Property** is maintained while allowing transactions to execute concurrently. (Isolation property maintained using locking)
  - To preserve database consistency by ensuring that the schedules of executing transactions are serializable.
  - To resolve read-write and write-write conflicts among transactions.

Concurrency Control Protocols (CCPs)

- A CCP is a **set of rules** enforced by the DBMS to ensure **serializable schedules**
- Also known as CCMs (Concurrency Control Methods)
- Main protocol known as **2PL (2-phase locking**), which is based on **locking the data items**

- Based on each transaction, transactions are securing a **lock** on a data item before using it:
  - Locking enforces **mutual exclusion** when accessing a data item – simplest kind of lock is a **binary lock,** which locks, only permission allowed to Read or Write a data item for a transaction. (**mutual exclusion** is a property of concurrency control which is used for preventing race condition among the concurrent execution of transaction)
    - Example: If T1 requests Lock(X) operation, the system grants the lock unless item X is already locked by another transaction. If request is granted, data item X is locked on behalf of the requesting transaction T1.
  - Unlocking operation removes the lock.
    - Example: If T1 issues Unlock (X), data item X is made available to all other transactions.

- System maintains **lock table** to **keep track of which items are locked** by which transactions
- Lock(X) and Unlock (X) are hence **system calls**
- Transactions that request a lock but do not have it granted can be placed on a **waiting queue** for the item
- Transaction T must unlock any items it had locked before T terminates

- Following code gives overview of lock and unlock operations

```
lock_item(X):
B:  if LOCK(X) = 0              (* item is unlocked *)
        then LOCK(X) ←1         (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 22.1
Lock and unlock operations for binary locks.

2PL Concurrency Control

- **Locking for database items:**

  For database purposes, *binary locks are not sufficient*:

  - Two locks modes are needed (a) **shared lock** (read lock) and (b) **exclusive lock** (write lock).

- **Shared mode**: Read lock (X). Several transactions can hold shared lock on X (because read operations are not conflicting).
- **Exclusive mode:** Write lock (X). Only one write lock on X can exist at any time on an item X. (No read or write locks on X by other transactions can exist).

Conflict matrix:

	Read	Write
Read	Y	N
Write	N	N

- Three operations are now needed:
  - read-lock(X): transaction T requests a read (shared) lock on item X
  - write-lock(X): transaction T requests a write (exclusive) lock on item X
  - unlock(X): transaction T unlocks an item that it holds a lock on (shared or exclusive)
- Transaction can be blocked (forced to wait) if the item is held by other transactions **in conflicting lock mode**

  - Conflicts are write-write or read-write (read-read is not conflicting)

The following are gives outline of these three operations

Two-Phase Locking Techniques: Essential components

- **Lock Manager:** Subsystem of DBMS that manages locks on data items.
- **Lock table:** Lock manager uses it to store information about *locked data items*, such as: data item id, transaction id, lock mode, list of waiting transaction ids, etc.
- One simple way to implement a lock table is through linked list (shown). Alternatively, a **hash table** with item id as hash key can be used.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

- **Rules for locking:**

- Transaction must request appropriate lock on a data item X before it reads or writes X.
- If T holds a write (exclusive) lock on X, it can both read and write X.
- If T holds a read lock on X, it can only read X.
- T must unlock all items that it holds before terminating
- (also T cannot unlock X unless it holds a lock on X).

Lock conversion

Lock upgrade: existing read lock to write lock

if Ti holds a read-lock on X, and no other Tj holds a read-lock on X (i □ j), then

it is possible to convert (**upgrade**) read-lock(X) to write-lock(X)

else

force Ti to wait until all other transactions Tj that hold read locks on X release their locks

Lock downgrade: existing write lock to read lock

if Ti holds a write-lock on X

(*this implies that no other transaction can have any lock on X*)

then it is possible to convert (**downgrade**) write-lock(X) to read-lock(X)

Two-Phase Locking Rule:

Each transaction should have **two phases**: (a) Locking (Growing) phase, and (b) Unlocking (Shrinking) Phase.

- **Locking (Growing) Phase**: A transaction applies locks (read or write) on desired data items one at a time. Can also try to upgrade a lock.
- **Unlocking (Shrinking) Phase**: A transaction unlocks its locked data items one at a time. Can also downgrade a lock.

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase no unlocking or downgrading of locks can occur, and during unlocking phase no new locking or upgrading operations are allowed.

Basic Two Phase Locking:

- When transaction starts executing, it is in the **locking phase,** and it can request locks on new items or upgrade locks. A transaction may be blocked (forced to wait) if a lock request is not granted. (This may lead to several transactions being in a *state of deadlock* – see later)
- Once the transaction unlocks an item (or downgrades a lock), it starts its **shrinking phase** and can no longer upgrade locks or request new locks.
- The combination of locking rules and 2-phase rule *ensures serializable schedules*

Theorem: If *every transaction* in a schedule follows the 2PL rules, the schedule must be serializable.

- **Timestamp based protocol**

Timestamp

A monotonically increasing identifier (e.g., an integer, or the system clock time when a transaction starts) indicating the start order of a transaction. A larger timestamp value indicates a more recently started transaction.

Timestamp of transaction T is denoted by **TS(T)**

Timestamp ordering CCM uses **the transaction timestamps to serialize the execution of concurrent transactions** – allows only *one equivalent serial order* based on the transaction timestamps

Instead of locks, systems keeps track of two values for each data item X:

- **Read-TS(X):** The largest timestamp among all the timestamps of transactions that have successfully read item X
- **Write-TS(X):** The largest timestamp among all the timestamps of transactions that have successfully written X

- When a transaction T requests to read or write an item X, TS(T) is compared with read-TS(X) and write-TS(X) to determine if request is *out-of-order*

Basic Timestamp Ordering

1. Transaction T requests a write-item(X) operation:

   If read-TS(X) > TS(T) or if write-TS(X) > TS(T), then a younger transaction has

   already read or written the data item so abort and roll-back T and reject the operation.

   Otherwise, execute write-item(X) of T and set write-TS(X) to TS(T).

2. Transaction T requests a read-item(X) operation:

   If write-TS(X) > TS(T), then a younger transaction has already written the data item X so abort and roll-back T and reject the operation.

   Otherwise, execute read-item(X) of T and set read-TS(X) to the larger of TS(T) and the current read-TS(X).

END OF UNIT- IV