*Fundamentals of*

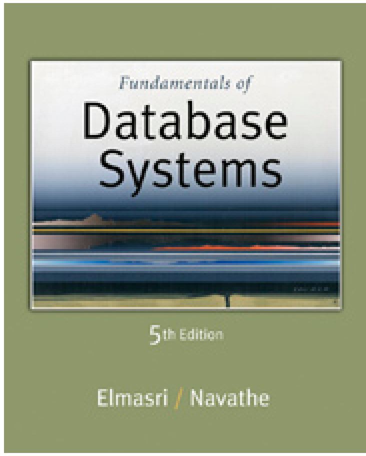
# Database Systems

5th Edition

Elmasri / Navathe

# Chapter 23

## Database Security and Authorization



# Chapter Outline

- 1 Database Security and Authorization
  - 1.1 Introduction to Database Security Issues
  - 1.2 Types of Security
  - 1.3 Database Security and DBA
  - 1.4 Access Protection, User Accounts, and Database Audits
- 2 Discretionary Access Control Based on Granting Revoking Privileges
  - 2.1 Types of Discretionary Privileges
  - 2.2 Specifying Privileges Using Views
  - 2.3 Revoking Privileges
  - 2.4 Propagation of Privileges Using the GRANT OPTION
  - 2.5 Specifying Limits on Propagation of Privileges

# Chapter Outline (contd.)

- 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security
  - 3.1 Comparing Discretionary Access Control and Mandatory Access Control
  - 3.2 Role-Based Access Control
  - 3.3 Access Control Policies for E-Commerce and the Web
- 4 Introduction to Statistical Database Security
- 5 Introduction to Flow Control
  - 5.1 Covert Channels
- 6 Encryption and Public Key Infrastructures
  - 6.1The Data and Advanced Encryption Standards
  - 6.2 Public Key Encryption
  - 6.3 Digital Signatures

# 1 Introduction to Database Security Issues

- Types of Security
  - Legal and ethical issues
  - Policy issues
  - System-related issues
  - The need to identify multiple security levels

# Introduction to Database Security Issues (2)

- Threats to databases
  - Loss of **integrity**
  - Loss of **availability**
  - Loss of **confidentiality**

- To protect databases against these types of threats four kinds of countermeasures can be implemented:
  - **Access control**
  - **Inference control**
  - **Flow control**
  - **Encryption**

# Introduction to Database Security Issues (3)

- A DBMS typically includes a database security and authorization subsystem that is responsible for ensuring the security portions of a database against unauthorized access.

- Two types of database security mechanisms:
  - **Discretionary** security mechanisms
  - **Mandatory** security mechanisms

# Introduction to Database Security Issues (4)

- The security mechanism of a DBMS must include provisions for restricting access to the database as a whole
  - This function is called **access control** and is handled by creating user accounts and passwords to control login process by the DBMS.

# Introduction to Database Security Issues (5)

- The security problem associated with databases is that of controlling the access to a **statistical database**, which is used to provide statistical information or summaries of values based on various criteria.

  - The countermeasures to **statistical database security** problem is called **inference control measures**.

# Introduction to Database Security Issues (6)

- Another security is that of **flow control**, which prevents information from flowing in such a way that it reaches unauthorized users.

- Channels that are pathways for information to flow implicitly in ways that violate the security policy of an organization are called **covert channels**.

# Introduction to Database Security Issues (7)

- A final security issue is **data encryption**, which is used to protect sensitive data (such as credit card numbers) that is being transmitted via some type communication network.
- The data is **encoded** using some **encoding algorithm**.
  - An unauthorized user who access encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher data.

# 1.2 Database Security and the DBA

- The database administrator (**DBA**) is the central authority for managing a database system.
  - The DBA's responsibilities include
    - granting privileges to users who need to use the system
    - classifying users and data in accordance with the policy of the organization
- The DBA is responsible for the overall security of the database system.

# 1.2 Database Security and the DBA (2)

- The DBA has a DBA account in the DBMS
  - Sometimes these are called a system or superuser account
  - These accounts provide powerful capabilities such as:
    - 1. Account creation
    - 2. Privilege granting
    - 3. Privilege revocation
    - 4. Security level assignment
  - Action 1 is access control, whereas 2 and 3 are discretionarym and 4 is used to control mandatory authorization

# 1.3 Access Protection, User Accounts, and Database Audits

- Whenever a person or group of person s need to access a database system, the individual or group must first apply for a user account.
  - The DBA will then create a new **account id** and **password** for the user if he/she deems there is a legitimate need to access the database
- The user must log in to the DBMS by entering account id and password whenever database access is needed.

# 1.3 Access Protection, User Accounts, and Database Audits(2)

- The database system must also keep **track of all operations** on the database that are applied by a certain user throughout **each login session**.
  - To keep a record of all updates applied to the database and of the particular user who applied each update, we can modify **system log**, which includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash.

# 1.3 Access Protection, User Accounts, and Database Audits(3)

- If any tampering with the database is suspected, a **database audit** is performed
  - A database audit consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period.
- A database log that is used mainly for security purposes is sometimes called an **audit trail**.

# Discretionary Access Control Based on Granting and Revoking Privileges

- The typical method of enforcing **discretionary access control** in a database system is based on the **granting** and **revoking privileges**.

# 2.1Types of Discretionary Privileges

- The **account level**:
  - At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- The **relation level** (or **table level**):
  - At this level, the DBA can control the privilege to access each individual relation or view in the database.

# 2.1Types of Discretionary Privileges(2)

- The privileges at the **account level** apply to the capabilities provided to the account itself and can include
  - the **CREATE SCHEMA** or **CREATE TABLE** privilege, to create a schema or base relation;
  - the **CREATE VIEW** privilege;
  - the **ALTER** privilege, to apply schema changes such adding or removing attributes from relations;
  - the **DROP** privilege, to delete relations or views;
  - the **MODIFY** privilege, to insert, delete, or update tuples;
  - and the **SELECT** privilege, to retrieve information from the database by using a **SELECT** query.

# 2.1Types of Discretionary Privileges(3)

- The second level of privileges applies to the **relation level**
  - This includes **base relations** and virtual (**view**) relations.
- The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the access matrix model where
  - The **rows** of a matrix M represents **subjects** (users, accounts, programs)
  - The **columns** represent **objects** (relations, records, columns, views, operations).
  - Each position **M(i,j)** in the matrix represents the types of privileges (read, write, update) that **subject i** holds on **object j**.

# 2.1Types of Discretionary Privileges(4)

- To control the granting and revoking of relation privileges, each relation R in a database is assigned and **owner account**, which is typically the account that was used when the relation was created in the first place.
  - The owner of a relation is given <u>all</u> privileges on that relation.
  - In SQL2, the DBA can assign and owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the **CREATE SCHEMA** command.
  - The owner account holder can **pass privileges** on any of the owned relation to other users by **granting** privileges to their accounts.

# 2.1Types of Discretionary Privileges(5)

- In SQL the following types of privileges can be granted on each individual relation R:
  - **SELECT** (retrieval or read) privilege on R:
    - Gives the account retrieval privilege.
    - In SQL this gives the account the privilege to use the **SELECT** statement to retrieve tuples from R.
  - **MODIFY** privileges on R:
    - This gives the account the capability to modify tuples of R.
    - In SQL this privilege is further divided into **UPDATE**, **DELETE**, and **INSERT** privileges to apply the corresponding SQL command to R.
    - In addition, both the **INSERT** and **UPDATE** privileges can specify that only certain attributes can be updated by the account.

# 2.1Types of Discretionary Privileges(6)

- In SQL the following types of privileges can be granted on each individual relation R (contd.):
  - **REFERENCES** privilege on R:
    - This gives the account the capability to **reference** relation R when specifying integrity constraints.
    - The privilege can also be **restricted** to specific attributes of R.

- Notice that to create a **view**, the account must have **SELECT** privilege on all relations involved in the view definition.

# 2.2 Specifying Privileges Using Views

- The mechanism of **views** is an important discretionary authorization mechanism in its own right. For example,
  - If the owner A of a relation R wants another account B to be able to <u>retrieve only some fields</u> of R, then A can create a view V of R that includes <u>only those attributes</u> and then grant SELECT on V to B.
  - The same applies to limiting B to retrieving <u>only certain tuples of</u> R; a view V' can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

# 2.3 Revoking Privileges

- In some cases it is desirable to grant a privilege to a user temporarily. For example,
  - The owner of a relation may want to grant the **SELECT** privilege to a user for a specific task and then revoke that privilege once the task is completed.
  - Hence, a mechanism for **revoking** privileges is needed. In SQL, a **REVOKE** command is included for the purpose of **canceling privileges**.

# 2.4 Propagation of Privileges using the GRANT OPTION

- Whenever the owner A of a relation R grants a privilege on R to another account B, privilege can be given to B with or without the **GRANT OPTION**.
- If the **GRANT OPTION** is given, this means that B can also grant that privilege on R to other accounts.
  - Suppose that B is given the **GRANT OPTION** by A and that B then grants the privilege on R to a third account C, also with **GRANT OPTION**. In this way, privileges on R can **propagate** to other accounts without the knowledge of the owner of R.
  - If the owner account <u>A now revokes</u> the privilege granted to B, <u>all the privileges that B propagated based</u> on that privilege should automatically <u>be revoked</u> by the system.

# 2.5 An Example

- Suppose that the DBA creates four accounts
  - A1, A2, A3, A4
- and wants only A1 to be able to create base relations. Then the DBA must issue the following GRANT command in SQL

```
GRANT CREATETAB TO A1;
```

- In SQL2 the same effect can be accomplished by having the DBA issue a **CREATE SCHEMA** command as follows:

```
CREATE SCHAMA EXAMPLE AUTHORIZATION A1;
```

# 2.5 An Example(2)

- User account A1 can create tables under the schema called **EXAMPLE**.
- Suppose that A1 **creates** the two base relations **EMPLOYEE** and **DEPARTMENT**
  - A1 is then **owner** of these two relations and hence all the relation privileges on each of them.
- Suppose that A1 wants to grant A2 the privilege to insert and delete tuples in both of these relations, but A1 does not want A2 to be able to propagate these privileges to additional accounts:

```
GRANT INSERT, DELETE ON
    EMPLOYEE, DEPARTMENT TO A2;
```

# 2.5 An Example(3)

**EMPLOYEE**

| Name | <u>Ssn</u> | Bdate | Address | Sex | Salary | Dno |
|---|---|---|---|---|---|---|

**DEPARTMENT**

| <u>Dnumber</u> | Dname | Mgr_ssn |
|---|---|---|

**Figure 23.1**
Schemas for the two relations EMPLOYEE and DEPARTMENT.

# 2.5 An Example(4)

- Suppose that A1 wants to allow A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts.
- A1 can issue the command:

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT
    TO A3 WITH GRANT OPTION;
```

- A3 can grant the **SELECT** privilege on the **EMPLOYEE** relation to A4 by issuing:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

  - Notice that A4 can’t propagate the SELECT privilege because GRANT OPTION was not given to A4

# 2.5 An Example(5)

- Suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 can issue:

  ```
  REVOKE SELECT ON EMPLOYEE FROM A3;
  ```

- The DBMS must now automatically revoke the SELECT privilege on EMPLOYEE from A4, too, because A3 granted that privilege to A4 and A3 does not have the privilege any more.

# 2.5 An Example(6)

- Suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege.
  - The limitation is to retrieve only the NAME, BDATE, and ADDRESS attributes and only for the tuples with DNO=5.
- A1 then create the view:

```
CREATE VIEW A3EMPLOYEE AS
  SELECT NAME, BDATE, ADDRESS
  FROM EMPLOYEE
  WHERE DNO = 5;
```

- After the view is created, A1 can grant **SELECT** on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3
    WITH GRANT OPTION;
```

# 2.5 An Example(7)

- Finally, suppose that A1 wants to allow A4 to update only the SALARY attribute of EMPLOYEE;
- A1 can issue:

```
GRANT UPDATE ON EMPLOYEE (SALARY) TO
A4;
```

  - The **UPDATE** or **INSERT** privilege can specify particular attributes that may be updated or inserted in a relation.
  - Other privileges (**SELECT**, **DELETE**) are not attribute specific.

# 2.6 Specifying Limits on Propagation of Privileges

- Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and are not a part of SQL.
  - Limiting **horizontal propagation** to an integer number i means that an account B given the GRANT OPTION can grant the privilege to at most i other accounts.
  - **Vertical propagation** is more complicated; it limits the depth of the granting of privileges.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security

- The discretionary access control techniques of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems.
- This is an all-or-nothing method:
  - A user either has or does not have a certain privilege.
- In many applications, and **additional security policy** is needed that classifies data and users based on security classes.
  - This approach as **mandatory access control**, would typically be **combined** with the discretionary access control mechanisms.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(2)

- Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest: TS ≥ S ≥ C ≥ U

- The commonly used model for multilevel security, known as the Bell-LaPadula model, classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications, T, S, C, or U:
  - **Clearance** (classification) of a subject S as **class(S**) and to the **classification** of an object O as **class(O)**.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(3)

- Two restrictions are enforced on data access based on the subject/object classifications:
  - **Simple security property:** A subject S is not allowed read access to an object O unless class(S) ≥ class(O).
  - A subject S is not allowed to write an object O unless class(S) ≤ class(O). This known as the **star property** (or * property).

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(4)

- To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects.
- Hence, each attribute A is associated with a **classification attribute C** in the schema, and each attribute value in a tuple is associated with a corresponding security classification.
- In addition, in some models, a **tuple classification** attribute TC is added to the relation attributes to provide a classification for each tuple as a whole.
- Hence, a **multilevel relation** schema R with n attributes would be represented as
  - $R(A_1,C_1,A_2,C_2, \ldots, A_n,C_n,TC)$
- where each Ci represents the classification attribute associated with attribute $A_i$.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(5)

- The value of the **TC** attribute in each tuple t – which is the highest of all attribute classification values within t – provides a general classification for the tuple itself, whereas each $C_i$ provides a finer security classification for each attribute value within the tuple.
  - The apparent key of a multilevel relation is the set of attributes that would have formed the primary key in a regular(single-level) relation.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(6)

- A multilevel relation will appear to contain different data to subjects (users) with different clearance levels.
  - In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower-level classification through a process known as **filtering**.
  - In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the **apparent key**.
- This leads to the concept of **polyinstantiation** where several tuples can have the same apparent key value but have different attribute values for users at different classification levels.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(7)

- In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the same security classification within each individual tuple.
- In addition, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key.
  - This **constraint** ensures that a user can see the key if the user is permitted to see any part of the tuple at all.

# 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security(8)

- Other integrity rules, called **null integrity** and **interinstance integrity**, informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

# 3.1 Comparing Discretionary Access Control and Mandatory Access Control

- **Discretionary Access Control (DAC)** policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains.
  - The main drawback of **DAC** models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs.

# 3.1 Comparing Discretionary Access Control and Mandatory Access Control(2)

- By contrast, mandatory policies ensure a high degree of protection in a way, they prevent any illegal flow of information.
- Mandatory policies have the drawback of being too rigid and they are only applicable in limited environments.
- In many practical situations, discretionary policies are preferred because they offer a better trade-off between security and applicability.

# 3.2 Role-Based Access Control

- **Role-based access control (RBAC)** emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprisewide systems.
- Its basic notion is that permissions are associated with roles, and users are assigned to appropriate roles.
- Roles can be created using the **CREATE ROLE** and **DESTROY ROLE** commands.
  - The **GRANT** and **REVOKE** commands discussed under DAC can then be used to assign and revoke privileges from roles.

# 3.2 Role-Based Access Control(2)

- **RBAC** appears to be a viable alternative to traditional discretionary and mandatory access controls; it ensures that only authorized users are given access to certain data or resources.
- Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.
- Role hierarchy in **RBAC** is a natural way of organizing roles to reflect the organization's lines of authority and responsibility.

# 3.2 Role-Based Access Control(3)

- Another important consideration in **RBAC** systems is the possible temporal constraints that may exist on roles, such as time and duration of role activations, and timed triggering of a role by an activation of another role.
- Using an **RBAC** model is highly desirable goal for addressing the key security requirements of Web-based applications.
- In contrast, discretionary access control (**DAC**) and mandatory access control (**MAC**) models **lack capabilities** needed to support the security requirements emerging enterprises and Web-based applications.

# 3.3 Access Control Policies for E-Commerce and the Web

- **E-Commerce environments** require elaborate policies that go beyond traditional DBMSs.
  - In an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience.
  - The access control mechanism should be flexible enough to support a wide spectrum of heterogeneous protection objects.
- A related requirement is the support for **content-based access-control**.

# 3.3 Access Control Policies for E-Commerce and the Web(2)

- Another requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications.
  - A possible solution, to better take into account user profiles in the formulation of access control policies, is to support the notion of credentials.
  - A **credential** is a set of properties concerning a user that are relevant for security purposes
    - For example, age, position within an organization
  - It is believed that the XML language can play a key role in access control for e-commerce applications.

# 4 Introduction to Statistical Database Security

- **Statistical databases** are used mainly to produce statistics on various populations.
- The database may contain **confidential data** on individuals, which should be protected from user access.
- Users are permitted to retrieve **statistical information** on the populations, such as **averages, sums, counts, maximums, minimums,** and **standard deviations**.

# 4 Introduction to Statistical Database Security(2)

- A **population** is a set of tuples of a relation (table) that satisfy some selection condition.
- Statistical queries involve applying **statistical functions** to a **population** of tuples.

# 4 Introduction to Statistical Database Security(3)

- For example, we may want to retrieve the *number* of individuals in a **population** or the *average income* in the population.
  - However, statistical users are not allowed to retrieve individual data, such as the income of a specific person.
- Statistical database security techniques must prohibit the retrieval of individual data.
- This can be achieved by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION.
  - Such queries are sometimes called **statistical queries**.

# 4 Introduction to Statistical Database Security(4)

- It is DBMS's responsibility to ensure confidentiality of information about individuals, while still providing useful statistical summaries of data about those individuals to users. Provision of **privacy protection** of users in a statistical database is paramount.
- In some cases it is possible to **infer** the values of individual tuples from a sequence statistical queries.
  - This is particularly true when the conditions result in a population consisting of a small number of tuples.

# 5 Introduction to Flow Control

- **Flow control** regulates the distribution or flow of information among accessible objects.
- A **flow** between object X and object Y occurs when a program reads values from X and writes values into Y.
  - Flow controls check that information contained in some objects does not flow explicitly or implicitly into less protected objects.
- A **flow policy** specifies the channels along which information is allowed to move.
  - The simplest flow policy specifies just two classes of information:
    - confidential (C) and nonconfidential (N)
  - and allows all flows except those from class C to class N.

# 5.1 Covert Channels

- A **covert channel** allows a transfer of information that violates the security or the policy.

- A **covert channel allows** information to pass from a higher classification level to a lower classification level through **improper means**.

# 5.1 Covert Channels(2)

- **Covert channels** can be classified into two broad categories:
  - **Storage channels** do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.
  - **Timing channel** allow the information to be conveyed by the timing of events or processes.
- Some security experts believe that one way to avoid covert channels is for programmers to not actually gain access to sensitive data that a program is supposed to process after the program has been put into operation.

# 6 Encryption and Public Key Infrastructures

- **Encryption** is a means of maintaining secure data in an insecure environment.
- **Encryption** consists of applying an **encryption algorithm** to data using some prespecified **encryption key**.
- The resulting data has to be **decrypted** using a **decryption key** to recover the original data.

# 6.1 The Data and Advanced Encryption Standards

- The **Data Encryption Standard (DES)** is a system developed by the U.S. government for use by the general public.
  - It has been widely accepted as a cryptographic standard both in the United States and abroad.
- **DES** can provide end-to-end encryption on the channel between the sender A and receiver B.

# 6.1 The Data and Advanced Encryption Standards(2)

- **DES** algorithm is a careful and complex combination of two of the fundamental building blocks of encryption:
  - **substitution** and **permutation** (transposition).
- The **DES** algorithm derives its strength from repeated application of these two techniques for a total of 16 cycles.
  - **Plaintext** (the original form of the message) is **encrypted** as blocks of **64 bits**.

# 6.1 The Data and Advanced Encryption Standards(3)

- After questioning the adequacy of **DES**, the National Institute of Standards (**NIST**) introduced the Advanced Encryption Standards (**AES**).
  - This algorithm has a block size of **128 bits** and thus takes longer time to crack.

# 6.2 Public Key Encryption

- In 1976 Diffie and Hellman proposed a new kind of cryptosystem, which they called **public key encryption**.
- **Public key algorithms** are based on **mathematical functions** rather than operations on bit patterns.
  - They also involve the use of **two separate keys**
    - in contrast to conventional encryption, which uses only one key.
  - The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication.

# 6.2 Public Key Encryption(2)

- The two keys used for public key encryption are referred to as the **public key** and the **private key**.
  - the **private key** is kept secret, but it is referred to as private key rather than a secret key (the word used in conventional encryption to avoid confusion with conventional encryption).

# 6.2 Public Key Encryption(3)

- A public key encryption scheme, or infrastructure, has six ingredients:
  - **Plaintext**: This is the data or readable message that is fed into the algorithm as input.
  - **Encryption algorithm**: The encryption algorithm performs various transformations on the **plaintext**.
  - **Public and private keys**: These are pair of keys that have been selected so that if one is used for encryption, the other is used for decryption.
    - The exec transformations performed by the encryption algorithm depend on the public or private key that is provided as input.

# 6.2 Public Key Encryption(4)

- A public key encryption scheme, or infrastructure, has six ingredients (contd.):
  - **Ciphertext**:
    - This is the scrambled message produced as output. It depends on the **plaintext** and the key.
    - For a given message, two different keys will produce two different **ciphertexts**.
  - **Decryption algorithm**:
    - This algorithm accepts the **ciphertext** and the matching key and produces the original **plaintext**.

# 6.2 Public Key Encryption(5)

- **Public** key is made for public and **private** key is known only by owner.
- A general-purpose public key cryptographic algorithm relies on
  - **one key for encryption** and
  - a different but related **key for decryption**.

# 6.2 Public Key Encryption(6)

- The essential steps are as follows:
  - Each user generates a **pair of keys** to be used for the encryption and decryption of messages.
  - Each user places one of the two keys in a public register or other accessible file. This is the **public key**. The companion key is kept private (**private key**).
  - If a sender wishes to send a private message to a receiver, the sender **encrypts** the message using the receiver's public key.
  - When the receiver receives the message, he or she **decrypts** it using the receiver's private key.
    - No other recipient can decrypt the message because only the receiver knows his or her private key.

# 6.2 Public Key Encryption(7)

- The **RSA Public Key Encryption** algorithm, one of the first public key schemes was introduced in 1978 by Ron Rivest (R), Adi Shamir (S), and Len Adleman (A) at MIT and is named after them.
  - The RSA encryption algorithm incorporates results from **number theory**, such as the difficulty of determining the large prime factors of a large number.
- The RSA algorithm also operates with **modular arithmetic** – mod n, where n is the product of two large prime numbers.

# 6.2 Public Key Encryption(8)

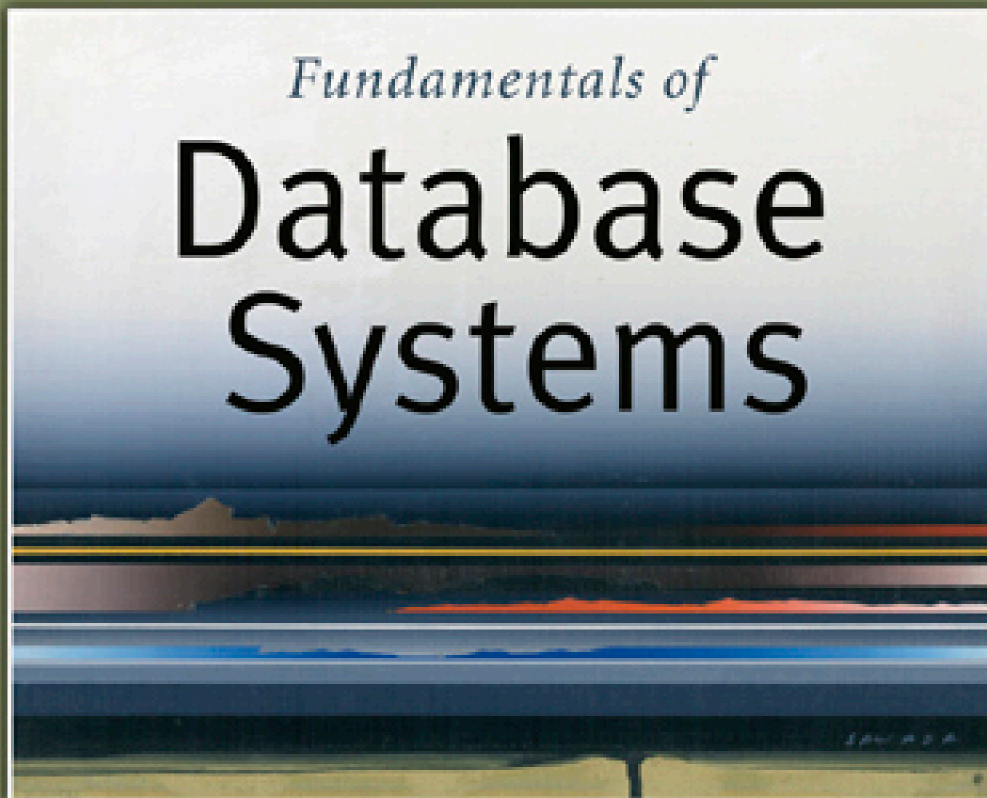
- Two keys, **d** and **e**, are used for decryption and encryption.
  - An important property is that **d** and **e** can be interchanged.
  - **n** is chosen as a large integer that is a product of **two large distinct prime numbers**, **a** and **b**.
  - The encryption key **e** is a randomly chosen number between 1 and **n** that is **relatively prime** to (a-1) x (b-1).
  - The **plaintext** block **P** is encrypted as **Pe mod n**.
  - Because the exponentiation is performed **mod n**, factoring **Pe** to uncover the encrypted plaintext is difficult.
  - However, the decryption key **d** is carefully chosen so that **(Pe)d mod n = P**.
  - The decryption key **d** can be computed from the condition that **d x e= 1 mod ((a-1)x(b-1))**.
  - Thus, the legitimate receiver who knows d simply computes **(Pe)d mod n = P** and recovers **P** without having to factor **Pe** .

# 6.3 Digital Signatures

- A **digital signature** is an example of using encryption techniques to provide authentication services in e-commerce applications.
- A digital signature is a means of associating a mark unique to an individual with a body of text.
  - The mark should be unforgettable, meaning that others should be able to check that the signature does come from the originator.
- A digital signature consists of a string of symbols.
  - Signature must be different for each use.
    - This can be achieved by making each digital signature a function of the message that it is signing, together with a time stamp.
  - Public key techniques are the means creating digital signatures.

# Summary

- 1 Database Security and Authorization
- 2 Discretionary Access Control
- 3 Mandatory Access Control and Role-Based Access Control for Multilevel Security
- 4 Statistical Database Security
- 5 Flow Control
- 6 Encryption and Public Key Infrastructures

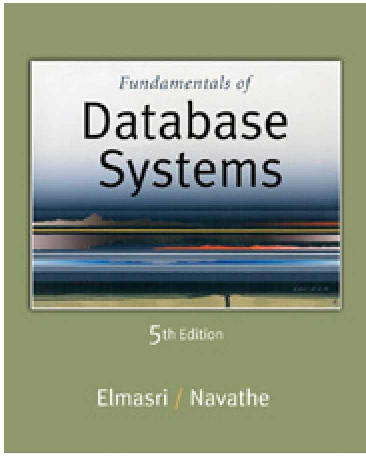



5th Edition

Elmasri / Navathe

# Chapter 18

## Concurrency Control Techniques



# Chapter 18 Outline

- Databases Concurrency Control
  1. Purpose of Concurrency Control
  2. Two-Phase locking
  3. Limitations of CCMs
  4. Index Locking
  5. Lock Compatibility Matrix
  6. Lock Granularity

# Database Concurrency Control

- 1 Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

# Database Concurrency Control

## Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Two locks modes:
  - (a) shared (read)        (b) exclusive (write).
- Shared mode:  shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

| | Read | Write |
|---|---|---|
| Read | Y | N |
| Write | N | N |

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

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

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Database requires that all transactions should be well-formed. A transaction is well-formed if:
  - It must lock the data item before it reads or writes to it.
  - It must not lock an already locked data items and it must not try to unlock a free data item.

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the lock operation:

```
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);
   goto B
   end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the unlock operation:

  LOCK (X) ← 0 (*unlock the item*)

  if any transactions are waiting then

    wake up one of the waiting the transactions;

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the read operation:

```
B: if LOCK (X) = “unlocked” then
 begin LOCK (X) ← “read-locked”;
    no_of_reads (X) ← 1;
 end
 else if LOCK (X) ← “read-locked” then
         no_of_reads (X) ← no_of_reads (X) +1
      else begin wait (until LOCK (X) = “unlocked” and
         the lock manager wakes up the transaction);
         go to B
      end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the write lock operation:

```
B: if LOCK (X) = "unlocked" then
 begin LOCK (X) ← "read-locked";
    no_of_reads (X) ← 1;
 end
 else if LOCK (X) ← "read-locked" then
         no_of_reads (X) ← no_of_reads (X) +1
      else begin wait (until LOCK (X) = "unlocked" and
        the lock manager wakes up the transaction);
        go to B
      end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the unlock operation:

```
if LOCK (X) = “write-locked” then
 begin LOCK (X) ← “unlocked”;
     wakes up one of the transactions, if any
 end
 else if LOCK (X) ← “read-locked” then
    begin
         no_of_reads (X) ← no_of_reads (X) -1
         if  no_of_reads (X) = 0 then
         begin
       LOCK (X) = “unlocked”;
      wake up one of the transactions, if any
         end
    end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Lock conversion
  - Lock upgrade: existing read lock to write lock

    if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then
     convert read-lock (X) to write-lock (X)
    else
     force Ti to wait until Tj unlocks X

  - Lock downgrade: existing write lock to read lock

    Ti has a write-lock (X)    (*no transaction can have any lock on X*)

    convert write-lock (X) to read-lock (X)

# Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two Phases:
  - (a) Locking (Growing)
  - (b) Unlocking (Shrinking).
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time.
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time.
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

| **T1** | **T2** | **Result** |
|---|---|---|
| read_lock (Y); | read_lock (X); | Initial values: X=20; Y=30 |
| read_item (Y); | read_item (X); | Result of serial execution |
| unlock (Y); | unlock (X); | T1 followed by T2 |
| write_lock (X); | Write_lock (Y); | X=50, Y=80. |
| read_item (X); | read_item (Y); | Result of serial execution |
| X:=X+Y; | Y:=X+Y; | T2 followed by T1 |
| write_item (X); | write_item (Y); | X=70, Y=50 |
| unlock (X); | unlock (Y); | |

## Two-Phase Locking Techniques: The algorithm

| T1 | T2 |
|---|---|
| read_lock (Y); | |
| read_item (Y); | |
| **unlock (Y);** | |
| | read_lock (X); |
| | read_item (X); |
| | **unlock (X);** |
| | **write_lock (Y);** |
| | read_item (Y); |
| | Y:=X+Y; |
| | write_item (Y); |
| | unlock (Y); |
| **write_lock (X);** | |
| read_item (X); | |
| X:=X+Y; | |
| write_item (X); | |
| unlock (X); | |

Time

**Result**

X=50; Y=50
Nonserializable because it.
violated two-phase policy.

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

| **T'1** | **T'2** | |
|---|---|---|
| read_lock (Y); | read_lock (X); | T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with. |
| read_item (Y); | read_item (X); | |
| write_lock (X); | Write_lock (Y); | |
| unlock (Y); | unlock (X); | |
| read_item (X); | read_item (Y); | |
| X:=X+Y; | Y:=X+Y; | |
| write_item (X); | write_item (Y); | |
| unlock (X); | unlock (Y); | |

# Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock**

| **T'1** | **T'2** | |
|---|---|---|
| read_lock (Y);<br>read_item (Y); | | T1 and T2 did follow two-phase policy but they are deadlock |
| | read_lock (X);<br>read_item (Y); | |
| write_lock (X);<br>(waits for X) | write_lock (Y);<br>(waits for Y) | |

- Deadlock (T'1 and T'2)

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock prevention**
  - A transaction locks all data items it refers to before it begins execution.
  - This way of locking prevents deadlock since a transaction never waits for a data item.
  - The conservative two-phase locking uses this approach.

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction o

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Starvation**
  - Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
  - This limitation is inherent in all priority based scheduling mechanisms.
  - In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

# Database Concurrency Control

Timestamp based concurrency control algorithm

- **Timestamp**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

# Database Concurrency Control

Timestamp based concurrency control algorithm

- **Basic Timestamp Ordering**
  - 1. Transaction T issues a write_item(X) operation:
    - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
    - If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).
  - 2. Transaction T issues a read_item(X) operation:
    - If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
    - If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

# Database Concurrency Control

Timestamp based concurrency control algorithm

- **Strict Timestamp Ordering**
  - 1. Transaction T issues a write_item(X) operation:
    - If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
  - 2. Transaction T issues a read_item(X) operation:
    - If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

# Database Concurrency Control

Timestamp based concurrency control algorithm

- **Thomas's Write Rule**
  - If read_TS(X) > TS(T) then abort and roll-back T and reject the operation.
  - If write_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
  - If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

# Database Concurrency Control

## Multiversion concurrency control techniques

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effect:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

# Database Concurrency Control

Multiversion technique based on timestamp ordering

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction.
  - Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effects: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

# Database Concurrency Control

## Multiversion technique based on timestamp ordering

- Assume X1, X2, …, Xn are the version of a data item X created by a write operation of transactions. With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.
- **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.
- **write_TS(Xi)**: The write timestamp of Xi that wrote the value of version Xi.
- A new version of Xi is created only by a write operation.

# Database Concurrency Control

## Multiversion technique based on timestamp ordering

- To ensure serializability, the following two rules are used.
- If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).
- If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

# Database Concurrency Control

## Multiversion technique based on timestamp ordering

- To ensure serializability, the following two rules are used.
  - If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).
  - If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).
- Rule 2 guarantees that a read will never be rejected.

# Database Concurrency Control

## Multiversion Two-Phase Locking Using Certify Locks

- Concept
  - Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.
  - This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X.

# Database Concurrency Control

## Multiversion Two-Phase Locking Using Certify Locks

- Steps
  1. X is the committed version of a data item.
  2. T creates a second version X' after obtaining a write lock on X.
  3. Other transactions continue to read X.
  4. T is ready to commit so it obtains a certify lock on X'.
  5. The committed version X becomes X'.
  6. T releases its certify lock on X', which is X now.

Compatibility tables for

| | Read | Write |
|---|---|---|
| Read | yes | no |
| Write | no | no |

read/write locking scheme

| | Read | Write | Certify |
|---|---|---|---|
| Read | yes | no | no |
| Write | no | no | no |
| Certify | no | no | no |

read/write/certify locking scheme

# Database Concurrency Control

## Multiversion Two-Phase Locking Using Certify Locks

- Note:
  - In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently.
  - This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes.  It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

# Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

- In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.
- Three phases:
  1. **Read phase**
  2. **Validation phase**
  3. **Write phase**

1. **Read phase**:
   - A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).

# Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

2. **Validation phase**: Serializability is checked before transactions write their updates to the database.

- This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
  - Tj completes its write phase before Ti starts its read phase.
  - Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj
  - Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase.
  - When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3 ) is checked. If none of these conditions holds, the validation fails and Ti is aborted.

# Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

3. **Write phase**: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

# Database Concurrency Control

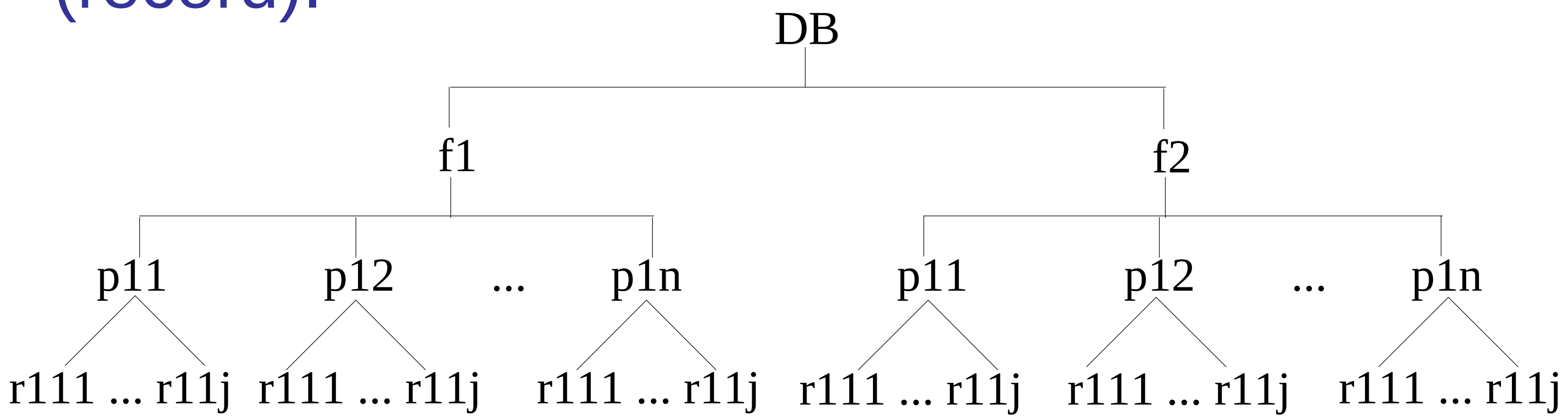
Granularity of data items and Multiple Granularity Locking

- A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation).
- Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity.
- Example of data item granularity:
  1. A field of a database record (an attribute of a tuple)
  2. A database record (a tuple or a relation)
  3. A disk block
  4. An entire file
  5. The entire database

# Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



# Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:
  - **Intention-shared (IS)**: indicates that a shared lock(s) will be requested on some descendent nodes(s).
  - **Intention-exclusive (IX)**: indicates that an exclusive lock(s) will be requested on some descendent node(s).
  - **Shared-intention-exclusive (SIX)**: indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

# Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- These locks are applied using the following compatibility matrix:

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | yes | yes | yes | yes | no |
| IX | yes | yes | no | no | no |
| S | yes | no | yes | no | no |
| SIX | yes | no | no | no | no |
| X | no | no | no | no | no |

Intention-shared (IS
Intention-exclusive (IX)
Shared-intention-exclusive (SIX)

# Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- The set of rules which must be followed for producing serializable schedule are
  1. The lock compatibility must adhered to.
  2. The root of the tree must be locked first, in any mode..
  3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
  4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
  5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
  6. T can unlock a node, N, only if none of the children of N are currently locked by T.

# Database Concurrency Control

Granularity of data items and Multiple Granularity Locking: An example of a serializable execution:

| T1 | T2 | T3 |
|---|---|---|
| IX(db) | | |
| IX(f1) | | |
| | IX(db) | |
| | | IS(db) |
| | | IS(f1) |
| | | IS(p11) |
| IX(p11) | | |
| X(r111) | | |
| | IX(f1) | |
| | X(p12) | |
| | | S(r11j) |
| IX(f2) | | |
| IX(p21) | | |
| IX(r211) | | |
| Unlock (r211) | | |
| Unlock (p21) | | |
| Unlock (f2) | | |
| | | S(f2) |

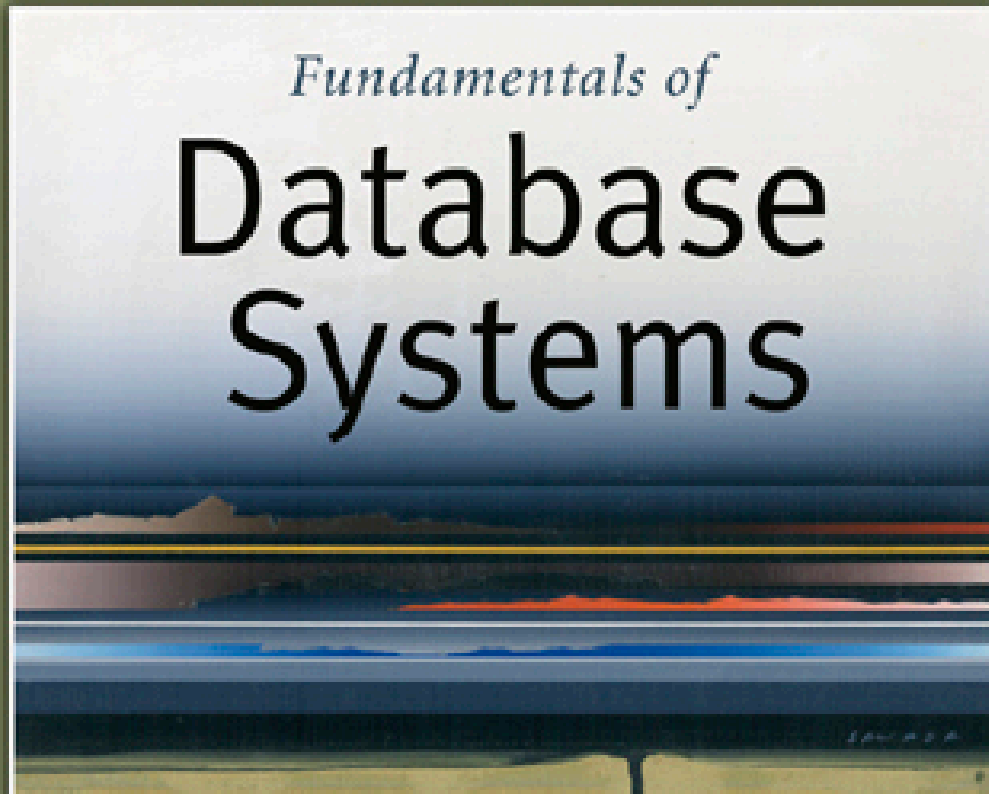
# Database Concurrency Control

- Granularity of data items and Multiple Granularity Locking: An example of a serializable execution (continued):

| T1 | T2 | T3 |
|---|---|---|
| | unlock(p12) | |
| | unlock(f1) | |
| | unlock(db) | |
| unlock(r111) | | |
| unlock(p11) | | |
| unlock(f1) | | |
| unlock(db) | | |
| | | unlock (r111j) |
| | | unlock (p11) |
| | | unlock (f1) |
| | | unlock(f2) |
| | | unlock(db) |

# Summary

- Databases Concurrency Control
  1. Purpose of Concurrency Control
  2. Two-Phase locking
  3. Limitations of CCMs
  4. Index Locking
  5. Lock Compatibility Matrix
  6. Lock Granularity

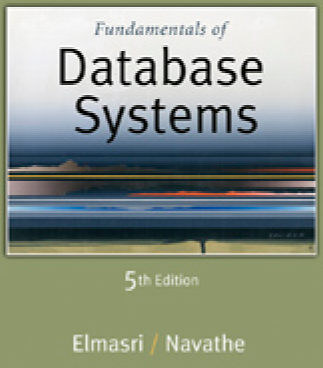



5th Edition

Elmasri / Navathe

# Chapter 17

## Introduction to Transaction Processing Concepts and Theory



# Chapter Outline

1 Introduction to Transaction Processing

2 Transaction and System Concepts

3 Desirable Properties of Transactions

4 Characterizing Schedules based on Recoverability

5 Characterizing Schedules based on Serializability

6 Transaction Support in SQL

# 1 Introduction to Transaction Processing (1)

- **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.

# Introduction to Transaction Processing (2)

- 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.

# Introduction to Transaction Processing (3)

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.

# Introduction to Transaction Processing (4)

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_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.

# Introduction to Transaction Processing (5)

READ AND WRITE OPERATIONS (contd.):

- **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

- FIGURE 17.2 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);
```

# Introduction to Transaction Processing (6)

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.
- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **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.

# Concurrent execution is uncontrolled: (a) The lost update problem.

**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.

**(a)**

Time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$); | |
| $X := X - N$; | |
| | read_item($X$); |
| | $X := X + M$; |
| write_item($X$); | |
| read_item($Y$); | |
| | write_item($X$); |
| $Y := Y + N$; | |
| write_item($Y$); | |

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

# Concurrent execution is uncontrolled: (b) The temporary update problem.

**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.

**(b)**

Time

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($Y$); | |

Transaction $T_1$ fails and must change the value of $X$ back to its old value; meanwhile $T_2$ has read the *temporary* incorrect value of $X$.

# Concurrent execution is uncontrolled: (c) The incorrect summary problem.

**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;<br>read_item($A$);<br>*sum* := *sum* + $A$;<br>⋮ |
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>*sum* := *sum* + $X$;<br>read_item($Y$);<br>*sum* := *sum* + $Y$; |
| read_item($Y$);<br>$Y := Y + N$;<br>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 $N$).

# Introduction to Transaction Processing (12)

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.

# Introduction to Transaction Processing (13)

Why **recovery** is needed (Contd.):
(What causes a Transaction to fail)

3. Local errors or exception conditions detected by the transaction:
   - Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. 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.
   - A programmed abort in the transaction causes it to fail.
4. Concurrency control enforcement:
   - The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (see Chapter 18).

# Introduction to Transaction Processing (14)

Why **recovery** is needed (contd.):

(What causes a Transaction to fail)

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 (1)

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - 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

# Transaction and System Concepts (2)

- 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.

# Transaction and System Concepts (3)

- Recovery manager keeps track of the following operations (cont):
  - **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.

# Transaction and System Concepts (4)

- 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.

# State transition diagram illustrating the states for transaction execution

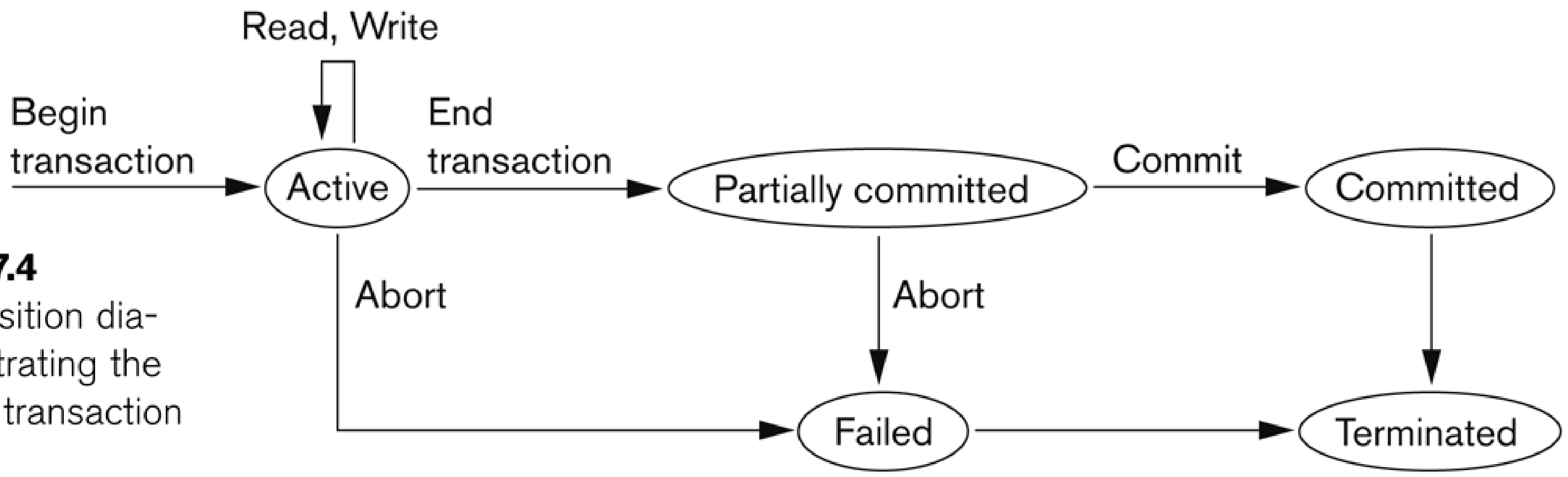



**Figure 17.4**
State transition diagram illustrating the states for transaction execution.

# Transaction and System Concepts (6)

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

# Transaction and System Concepts (7)

- The System Log (cont):
  - 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]: Records that transaction T has started execution.
    - [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
    - [read_item,T,X]: Records that transaction T has read the value of database item X.
    - [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - [abort,T]: Records that transaction T has been aborted.

# Transaction and System Concepts (8)

- The System Log (cont):
  - 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 (see Section 17.4).

# Transaction and System Concepts (9)

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 described in Chapter 19.
  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.

# Transaction and System Concepts (10)

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.

# Transaction and System Concepts (11)

Commit Point of a Transaction (cont):

- **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.

# 3 Desirable Properties of Transactions (1)

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 (see Chapter 21).
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

# 4 Characterizing Schedules based on Recoverability (1)

- **Transaction schedule or history**:
  - When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

# Characterizing Schedules based on Recoverability (2)

Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
- **Cascadeless schedule**:
  - One where every transaction reads only the items that are written by committed transactions.

# Characterizing Schedules based on Recoverability (3)

Schedules classified on recoverability (contd.):

- **Schedules requiring cascaded rollback**:
  - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Strict Schedules**:
  - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

# 5 Characterizing Schedules based on Serializability (1)

- Serial schedule:
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
    - Otherwise, the schedule is called nonserial schedule.
- Serializable schedule:
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

# Characterizing Schedules based on Serializability (2)

- Result equivalent:
  - Two schedules are called result equivalent if they produce the same final state of the database.
- Conflict equivalent:
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- Conflict serializable:
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

# Characterizing Schedules based on Serializability (3)

- Being serializable is <u>not</u> the same as being serial
- Being serializable implies that the schedule is a <u>correct</u> schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

# Characterizing Schedules based on Serializability (4)

- Serializability is hard to check.
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

# Characterizing Schedules based on Serializability (5)

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends.
  - Hence, we reduce the problem of checking the whole schedule to checking only a **committed project** of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with two phase locking

# Characterizing Schedules based on Serializability (6)

- View equivalence:
  - A less restrictive definition of equivalence of schedules
- View serializability:
  - Definition of serializability based on view equivalence.
  - A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

# Characterizing Schedules based on Serializability (7)

- Two schedules are said to be view equivalent if the following three conditions hold:
  1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
  2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
  3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

# Characterizing Schedules based on Serializability (8)

- The premise behind view equivalence:
  - As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.
  - "**The view**": the read operations are said to see *the same view* in both schedules.

# Characterizing Schedules based on Serializability (9)

- **Relationship between view and conflict equivalence**:
  - The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
  - Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.
  - Any conflict serializable schedule is also view serializable, but not vice versa.

# Characterizing Schedules based on Serializability (10)

- Relationship between view and conflict equivalence (cont):
  - Consider the following schedule of three transactions
    - T1: r1(X), w1(X);    T2: w2(X);    and    T3: w3(X):
  - Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

- In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.
  - Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3.
  - However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule.

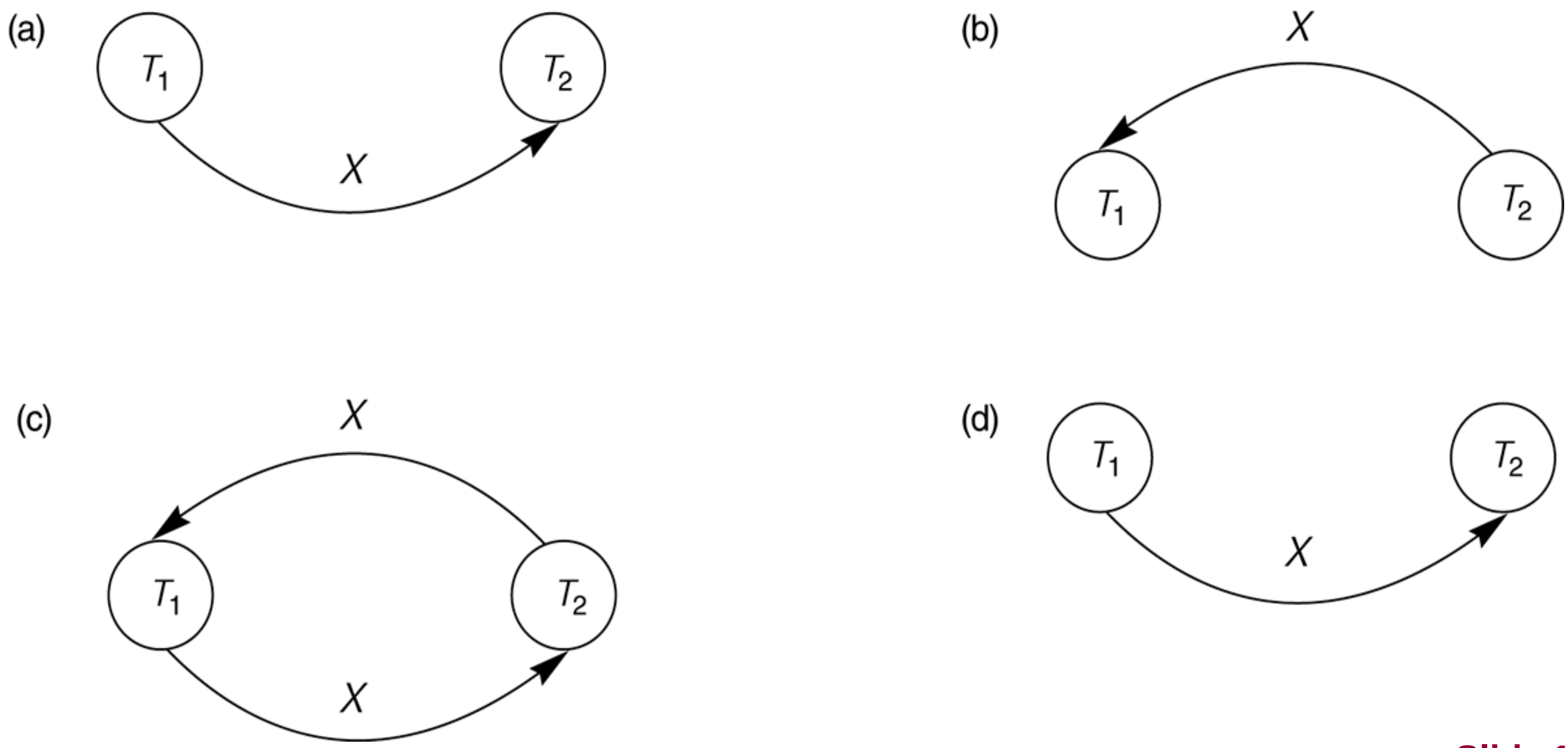
# Characterizing Schedules based on Serializability (11)

**Testing for conflict serializability: Algorithm 17.1:**

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

# Constructing the Precedence Graphs

- FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.
  - (a) Precedence graph for serial schedule A.
  - (b) Precedence graph for serial schedule B.
  - (c) Precedence graph for schedule C (not serializable).
  - (d) Precedence graph for schedule D (serializable, equivalent to schedule A).



# Another example of serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule F.

**(a)**

| Transaction $T_1$ |
|---|
| read_item($X$); |
| write_item($X$); |
| read_item($Y$); |
| write_item($Y$); |

| Transaction $T_2$ |
|---|
| read_item($Z$); |
| read_item($Y$); |
| write_item($Y$); |
| read_item($X$); |
| write_item($X$); |

| Transaction $T_3$ |
|---|
| read_item($Y$); |
| read_item($Z$); |
| write_item($Y$); |
| write_item($Z$); |

# Another Example of Serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule F.

**(b)**

Time ↓

| Transaction $T_1$ | Transaction $T_2$ | Transaction $T_3$ |
|---|---|---|
| | read_item($Z$); | |
| | 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**

# Another Example of Serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule *F*.

**(c)**

Time ↓

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

**Schedule F**

# Characterizing Schedules based on Serializability (14)

## Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly.
  - Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

# Characterizing Schedules based on Serializability (15)

Other Types of Equivalence of Schedules (contd.)

- Example: bank credit / debit transactions on a given item are **separable** and **commutative**.
  - Consider the following schedule S for the two transactions:
  - Sh : r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);
  - Using conflict serializability, it is **not serializable**.
  - However, if it came from a (read,update, write) sequence as follows:
    - r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20;r1(Y);
    - Y := Y + 10; w1(Y); r2(X); X := X + 20; (X);
  - Sequence explanation: debit, debit, credit, credit.
  - It is a *correct schedule* *for the given semantics*

# 6 Transaction Support in SQL2 (1)

- 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 <u>no explicit Begin</u> Transaction statement.
  - Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction <u>must have an explicit end</u> statement, which is either a COMMIT or ROLLBACK.

# Transaction Support in SQL2 (2)

Characteristics specified by a SET TRANSACTION statement in SQL2:

- **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 in the diagnostic area.

# Transaction Support in SQL2 (3)

Characteristics specified by a SET TRANSACTION statement in SQL2 (contd.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
  - With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
  - However, if any transaction executes at a lower level, then serializability may be violated.

# Transaction Support in SQL2 (4)

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.

# Transaction Support in SQL2 (5)

- Potential problem with lower isolation levels (contd.):
  - 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.

# Transaction Support in SQL2 (6)

- Sample SQL transaction:

```
 EXEC SQL whenever sqlerror go to UNDO;
 EXEC SQL SET TRANSACTION
             READ WRITE
             DIAGNOSTICS 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:  ...
```

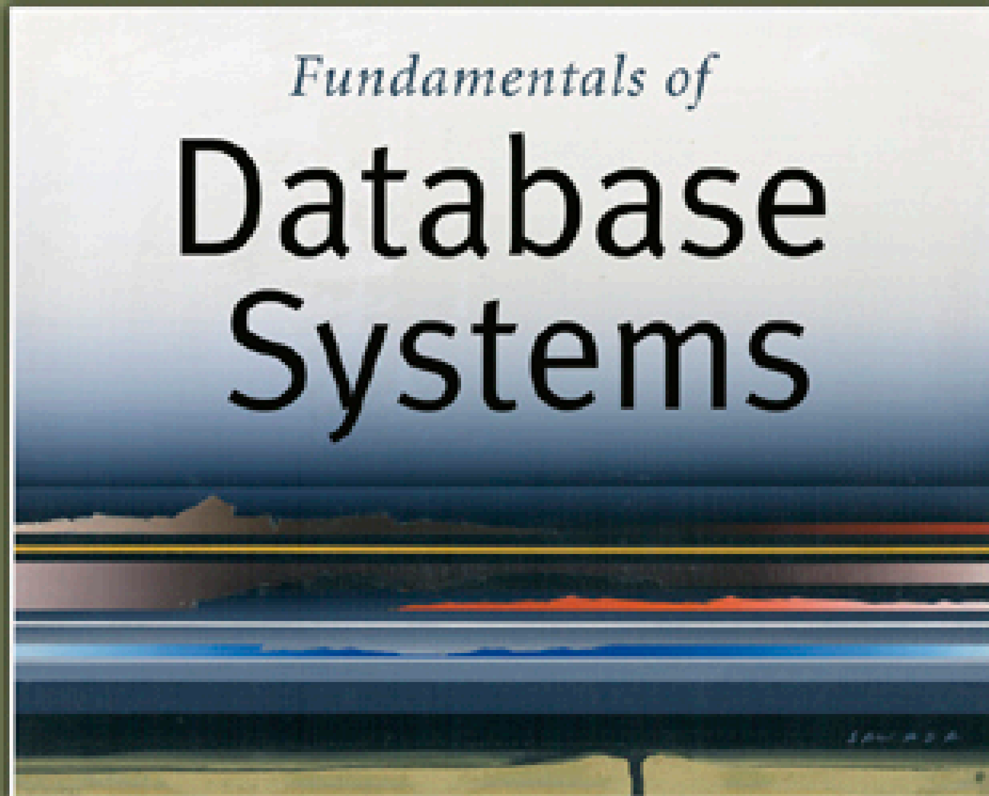
# Transaction Support in SQL2 (7)

- 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 |

# Summary

- Transaction and System Concepts
- Desirable Properties of Transactions
- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability
- Transaction Support in SQL

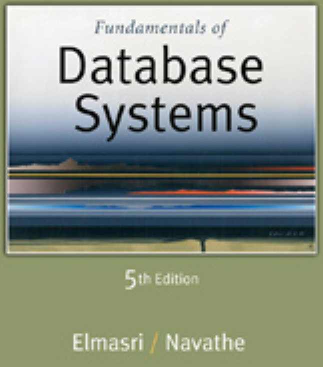



5th Edition

Elmasri / Navathe

# Chapter 19

## Database Recovery Techniques



# Chapter 19 Outline

## Databases Recovery

1. Purpose of Database Recovery
2. Types of Failure
3. Transaction Log
4. Data Updates
5. Data Caching
6. Transaction Roll-back (Undo) and Roll-Forward
7. Checkpointing
8. Recovery schemes
9. ARIES Recovery Scheme
10. Recovery in Multidatabase System

# Database Recovery

1 Purpose of Database Recovery
  - To bring the database into the last consistent state, which existed prior to the failure.
  - To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).
- Example:
  - If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

# Database Recovery

## 2 Types of Failure

- The database may become unavailable for use due to
  - **Transaction failure**: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
  - **System failure**: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
  - **Media failure**: Disk head crash, power disruption, etc.

# Database Recovery

## 3 Transaction Log

- For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.
- These values and other information is stored in a sequential file called Transaction log. A sample log is given below. Back P and Next P point to the previous and next log records of the same transaction.

| T ID | Back P | Next P | Operation | Data item | BFIM | AFIM |
|---|---|---|---|---|---|---|
| T1 | 0 | 1 | Begin | | | |
| T1 | 1 | 4 | Write | X | X = 100 | X = 200 |
| T2 | 0 | 8 | Begin | | | |
| T1 | 2 | 5 | W | Y | Y = 50 | Y = 100 |
| T1 | 4 | 7 | R | M | M = 200 | M = 200 |
| T3 | 0 | 9 | R | N | N = 400 | N = 400 |
| T1 | 5 | nil | End | | | |

# Database Recovery

4 Data Update

- **Immediate Update**: As soon as a data item is modified in cache, the disk copy is updated.
- **Deferred Update**: All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- **Shadow update**: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- **In-place update**: The disk version of the data item is overwritten by the cache version.

# Database Recovery

## 5 Data Caching

- Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk.
- The flushing is controlled by **Modified** and **Pin-Unpin** bits.
  - **Pin-Unpin**: Instructs the operating system not to flush the data item.
  - **Modified**: Indicates the AFIM of the data item.

# Database Recovery

6 Transaction **Roll-back (Undo)** and **Roll-Forward (Redo)**

- To maintain atomicity, a transaction's operations are redone or undone.
  - **Undo**: Restore all BFIMs on to disk (Remove all AFIMs).
  - **Redo**: Restore all AFIMs on to disk.
- Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.

# Database Recovery

**(a)**

| $T_1$ |
|---|
| read_item($A$) |
| read_item($D$) |
| write_item($D$) |

| $T_2$ |
|---|
| read_item($B$) |
| write_item($B$) |
| read_item($D$) |
| write_item($D$) |

| $T_3$ |
|---|
| read_item($C$) |
| write_item($B$) |
| read_item($A$) |
| write_item($A$) |

**Figure 19.1**
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

# Database Recovery

**(b)**

| | | A | B | C | D |
|---|---|---|---|---|---|
| | | 30 | 15 | 40 | 20 |
| | [start_transaction, $T_3$] | | | | |
| | [read_item, $T_3$, $C$] | | | | |
| * | [write_item, $T_3$, $B$, 15, 12] | | 12 | | |
| | [start_transaction, $T_2$] | | | | |
| | [read_item, $T_2$, $B$] | | | | |
| ** | [write_item, $T_2$, $B$, 12, 18] | | 18 | | |
| | [start_transaction, $T_1$] | | | | |
| | [read_item, $T_1$, $A$] | | | | |
| | [read_item, $T_1$, $D$] | | | | |
| | [write_item, $T_1$, $D$, 20, 25] | | | | 25 |
| | [read_item, $T_2$, $D$] | | | | |
| ** | [write_item, $T_2$, $D$, 25, 26] | | | | 26 |
| | [read_item, $T_3$, $A$] | | | | |

← System crash

* $T_3$ is rolled back because it did not reach its commit point.

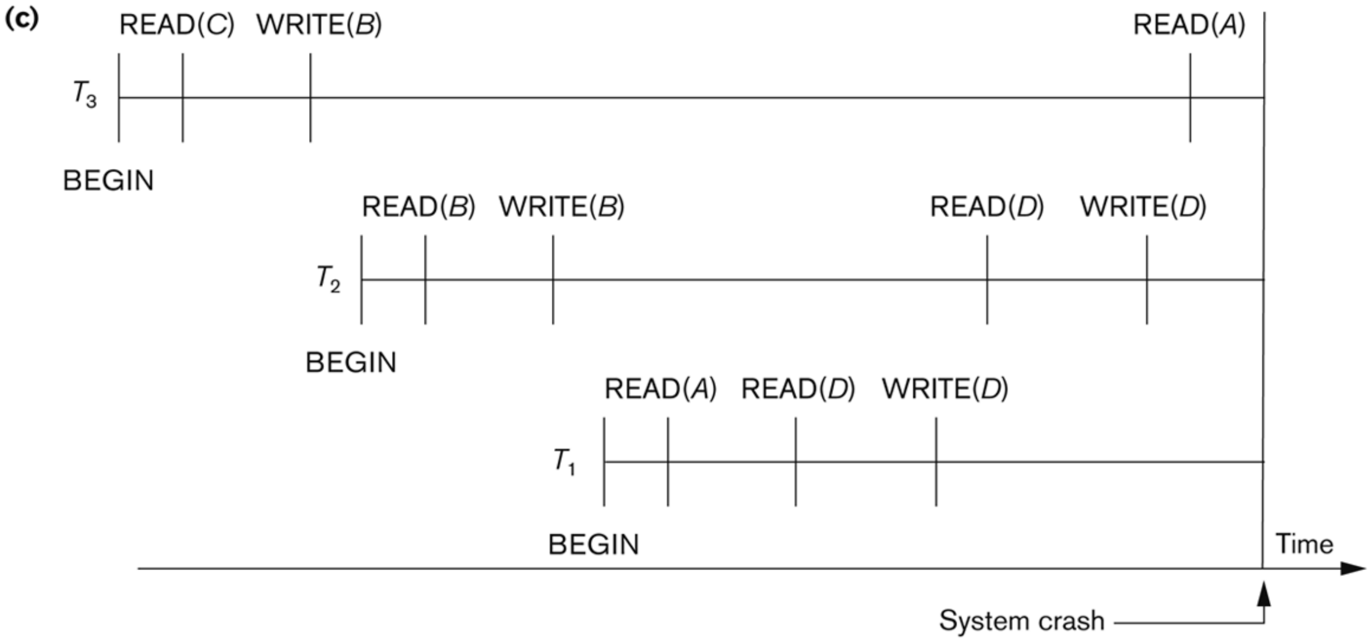
** $T_2$ is rolled back because it reads the value of item $B$ written by $T_3$.

**Figure 19.1**
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

# Database Recovery

**Roll-back**: One execution of T1, T2 and T3 as recorded in the log.

**Figure 19.1**
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.



# Database Recovery

Write-Ahead Logging

- When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging (WAL)** protocol. WAL states that
  - **For Undo**: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).
  - **For Redo**: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

# Database Recovery

7 Checkpointing

- Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps defines a checkpoint operation:
  1. Suspend execution of transactions temporarily.
  2. Force write modified buffer data to disk.
  3. Write a [checkpoint] record to the log, save the log to disk.
  4. Resume normal transaction execution.
- During recovery redo or undo is required to transactions appearing after [checkpoint] record.

# Database Recovery

## Steal/No-Steal and Force/No-Force

- Possible ways for flushing database cache to database disk:
  1. Steal: Cache can be flushed before transaction commits.
  2. No-Steal: Cache cannot be flushed before transaction commit.
  3. Force: Cache is immediately flushed (forced) to disk.
  4. No-Force: Cache is deferred until transaction commits
- These give rise to four different ways for handling recovery:
  - Steal/No-Force (Undo/Redo)
  - Steal/Force (Undo/No-redo)
  - No-Steal/No-Force (Redo/No-undo)
  - No-Steal/Force (No-undo/No-redo)

# Database Recovery

## 8 Recovery Scheme

- Deferred Update (No Undo/Redo)
  - The data update goes as follows:
  - A set of transactions records their updates in the log.
  - At commit point under WAL scheme these updates are saved on database disk.
  - After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

# Database Recovery

- Deferred Update in a single-user system
  There is no concurrent data sharing in a single user system. The data update goes as follows:
  - A set of transactions records their updates in the log.
  - At commit point under WAL scheme these updates are saved on database disk.
- After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

# Database Recovery

**(a)**

| $T_1$ |
|---|
| read_item($A$) |
| read_item($D$) |
| write_item($D$) |

| $T_2$ |
|---|
| read_item($B$) |
| write_item($B$) |
| read_item($D$) |
| write_item($D$) |

**(b)**

| | |
|---|---|
| [start_transaction, $T_1$] | |
| [write_item, $T_1$, $D$, 20] | |
| [commit, $T_1$] | |
| [start_transaction, $T_2$] | |
| [write_item, $T_2$, $B$, 10] | |
| [write_item, $T_2$, $D$, 25] | ← System crash |

The [write_item,...] operations of $T_1$ are redone.

$T_2$ log entries are ignored by the recovery process.

**Figure 19.2**
An example of recovery using deferred update in a single-user environment. (a) The READ and WRITE operations of two transactions. (b) The system log at the point of crash.

# Database Recovery

Deferred Update with concurrent users

- This environment requires some concurrency control mechanism to guarantee **isolation** property of transactions. In a system recovery transactions which were recorded in the log after the last checkpoint were **redone**. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.

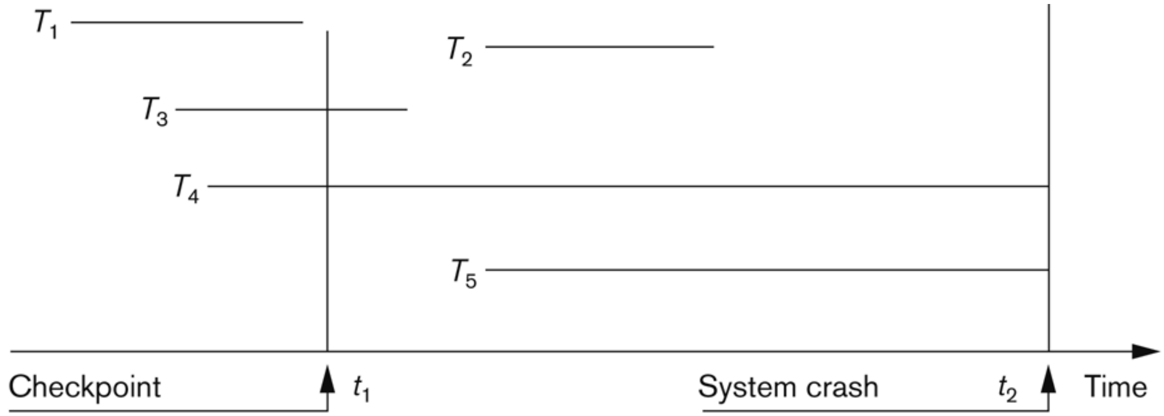



**Figure 19.3**
An example of recovery in a multi-user environment

# Database Recovery

**(a)**

| $T_1$ |
|---|
| read_item($A$) |
| read_item($D$) |
| write_item($D$) |

| $T_2$ |
|---|
| read_item($B$) |
| write_item($B$) |
| read_item($D$) |
| write_item($D$) |

| $T_3$ |
|---|
| read_item($A$) |
| write_item($A$) |
| read_item($C$) |
| write_item($C$) |

| $T_4$ |
|---|
| read_item($B$) |
| write_item($B$) |
| read_item($A$) |
| write_item($A$) |

**(b)**

| | |
|---|---|
| [start_transaction, $T_1$] | |
| [write_item, $T_1$, $D$, 20] | |
| [commit, $T_1$] | |
| [checkpoint] | |
| [start_transaction, $T_4$] | |
| [write_item, $T_4$, $B$, 15] | |
| [write_item, $T_4$, $A$, 20] | |
| [commit, $T_4$] | |
| [start_transaction, $T_2$] | |
| [write_item, $T_2$, $B$, 12] | |
| [start_transaction, $T_3$] | |
| [write_item, $T_3$, $A$, 30] | |
| [write_item, $T_2$, $D$, 25] | ← System crash |

$T_2$ and $T_3$ are ignored because they did not reach their commit points.

$T_4$ is redone because its commit point is after the last system checkpoint.

**Figure 19.4**
An example of recovery using deferred update with concurrent transactions.
(a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

# Database Recovery

Deferred Update with concurrent users

- Two tables are required for implementing this protocol:
  - **Active table**: All active transactions are entered in this table.
  - **Commit table**: Transactions to be committed are entered in this table.

- During recovery, all transactions of the **commit** table are redone and all transactions of **active** tables are ignored since none of their AFIMs reached the database. It is possible that a **commit** table transaction may be **redone** twice but this does not create any inconsistency because of a redone is "**idempotent**", that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

# Database Recovery

## Recovery Techniques Based on Immediate Update

- **Undo/No-redo Algorithm**
  - In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits.
  - For this reason the recovery manager **undoes** all transactions during recovery.
  - No transaction is **redone**.
  - It is possible that a transaction might have completed execution and ready to commit but this transaction is also **undone**.

# Database Recovery

Recovery Techniques Based on Immediate Update

- **Undo/Redo Algorithm** (**Single-user** environment)
  - Recovery schemes of this category apply **undo** and also **redo** for recovery.
  - In a single-user environment no concurrency control is required but a log is maintained under WAL.
  - Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table.
  - The recovery manager performs:
    - **Undo** of a transaction if it is in the **active** table.
    - **Redo** of a transaction if it is in the **commit** table.

# Database Recovery

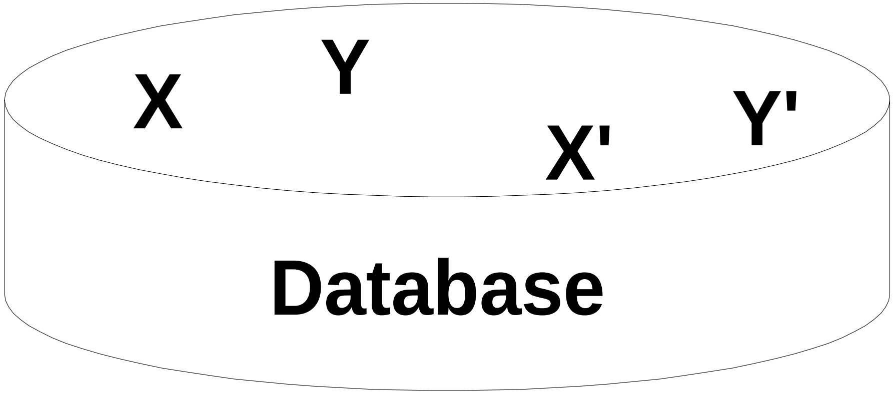
Recovery Techniques Based on Immediate Update

- **Undo/Redo Algorithm** (**Concurrent** execution)
- Recovery schemes of this category applies **undo** and also **redo** to recover the database from failure.
- In concurrent execution environment a concurrency control is required and log is maintained under WAL.
- Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager checkpointing is used.
- The recovery performs:
  - **Undo** of a transaction if it is in the **active** table.
  - **Redo** of a transaction if it is in the **commit** table.

# Database Recovery

## Shadow Paging

- The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.




X and Y: Shadow copies of data items

X' and Y': Current copies of data items

# Database Recovery

## Shadow Paging

- To manage access of data items by concurrent transactions two directories (current and shadow) are used.
  - The directory arrangement is illustrated below. Here a page is a data item.

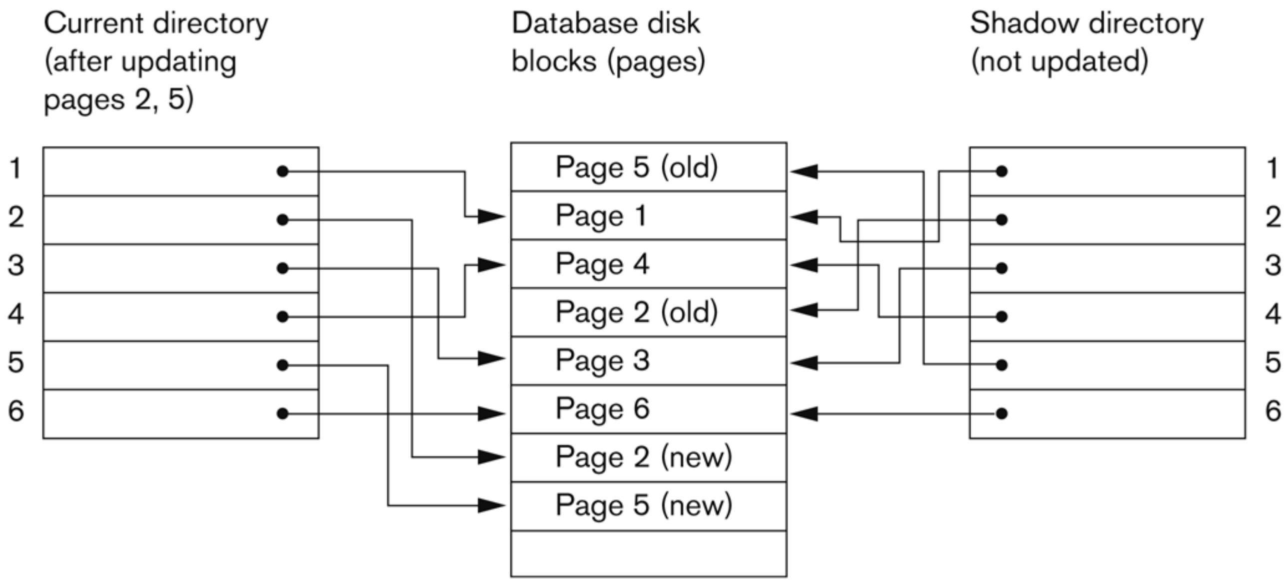



**Figure 19.5**
An example of shadow paging.

# Database Recovery

## The ARIES Recovery Algorithm

- The ARIES Recovery Algorithm is based on:
  - **WAL** (Write Ahead Logging)
  - **Repeating history during redo**:
    - ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred.
  - **Logging changes during undo**:
    - It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

# Database Recovery

**The ARIES Recovery Algorithm (contd.)**

- The ARIES recovery algorithm consists of three steps:
  1. **Analysis**: step identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of crash. The appropriate point in the log where redo is to start is also determined.
  2. **Redo**: necessary redo operations are applied.
  3. **Undo**: log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.

# Database Recovery

**The ARIES Recovery Algorithm (contd.)**

- **The Log and Log Sequence Number (LSN)**
  - A log record is written for:
    - (a) data update
    - (b) transaction commit
    - (c) transaction abort
    - (d) undo
    - (e) transaction end
  - In the case of undo a compensating log record is written.

# Database Recovery

## The ARIES Recovery Algorithm (contd.)

- **The Log and Log Sequence Number (LSN) (contd.)**
  - A unique LSN is associated with every log record.
    - LSN increases monotonically and indicates the disk address of the log record it is associated with.
    - In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.
  - A log record stores
    - (a) the previous LSN of that transaction
    - (b) the transaction ID
    - (c) the type of log record.

# Database Recovery

**The ARIES Recovery Algorithm (contd.)**

- **The Log and Log Sequence Number (LSN) (contd.)**
- A log record stores:
  1. Previous LSN of that transaction: It links the log record of each transaction. It is like a back pointer points to the previous record of the same transaction
  2. Transaction ID
  3. Type of log record
- For a write operation the following additional information is logged:
  1. Page ID for the page that includes the item
  2. Length of the updated item
  3. Its offset from the beginning of the page
  4. BFIM of the item
  5. AFIM of the item

# Database Recovery

## The ARIES Recovery Algorithm (contd.)

- The **Transaction table** and the **Dirty Page table**
  - For efficient recovery following tables are also stored in the log during checkpointing:
    - **Transaction table**: Contains an entry for each active transaction, with information such as transaction ID, transaction status and the LSN of the most recent log record for the transaction.
    - **Dirty Page table**: Contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

# Database Recovery

## The ARIES Recovery Algorithm (contd.)

- Checkpointing
  - A checkpointing does the following:
    - Writes a begin_checkpoint record in the log
    - Writes an end_checkpoint record in the log. With this record the contents of transaction table and dirty page table are appended to the end of the log.
    - Writes the LSN of the begin_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information.
  - To reduce the cost of checkpointing and allow the system to continue to execute transactions, ARIES uses "fuzzy checkpointing".

# Database Recovery

**The ARIES Recovery Algorithm (contd.)**

- The following steps are performed for recovery
  - **Analysis phase**: Start at the begin_checkpoint record and proceed to the end_checkpoint record. Access transaction table and dirty page table are appended to the end of the log. Note that during this phase some other log records may be written to the log and transaction table may be modified. The analysis phase compiles the set of redo and undo to be performed and ends.
  - **Redo phase**: Starts from the point in the log up to where all dirty pages have been flushed, and move forward to the end of the log. Any change that appears in the dirty page table is redone.
  - **Undo phase**: Starts from the end of the log and proceeds backward while performing appropriate undo. For each undo it writes a compensating record in the log.
- The recovery completes at the end of undo phase.

# Database Recovery

(a)

| Lsn | Last_lsn | Tran_id | Type | Page_id | Other_information |
|---|---|---|---|---|---|
| 1 | 0 | $T_1$ | update | *C* | . . . |
| 2 | 0 | $T_2$ | update | *B* | . . . |
| 3 | 1 | $T_1$ | commit | | . . . |
| 4 | begin checkpoint | | | | |
| 5 | end checkpoint | | | | |
| 6 | 0 | $T_3$ | update | *A* | . . . |
| 7 | 2 | $T_2$ | update | *C* | . . . |
| 8 | 7 | $T_2$ | commit | | . . . |

(b)

**TRANSACTION TABLE**

| Transaction_id | Last_lsn | Status |
|---|---|---|
| $T_1$ | 3 | commit |
| $T_2$ | 2 | in progress |

**DIRTY PAGE TABLE**

| Page_id | Lsn |
|---|---|
| *C* | 1 |
| *B* | 2 |

(c)

**TRANSACTION TABLE**

| Transaction_id | Last_lsn | Status |
|---|---|---|
| $T_1$ | 3 | commit |
| $T_2$ | 8 | commit |
| $T_3$ | 6 | in progress |

**DIRTY PAGE TABLE**

| Page_id | Lsn |
|---|---|
| *C* | 1 |
| *B* | 2 |
| *A* | 6 |

**Figure 19.6**
An example of recovery in ARIES. (a) The log at point of crash.
(b) The Transaction and Dirty Page Tables at time of checkpoint.
(c) The Transaction and Dirty Page Tables after the analysis phase.

# Database Recovery

10 Recovery in multidatabase system

- A multidatabase system is a special distributed database system where one node may be running relational database system under UNIX, another may be running object-oriented system under Windows and so on.
- A transaction may run in a distributed fashion at multiple nodes.
- In this execution scenario the transaction commits only when all these multiple nodes agree to commit individually the part of the transaction they were executing.
- This commit scheme is referred to as "**two-phase commit**" (**2PC**).
  - If any one of these nodes fails or cannot commit the part of the transaction, then the transaction is aborted.
- Each node recovers the transaction under its own recovery protocol.

# Summary

- Databases Recovery
  - Types of Failure
  - Transaction Log
  - Data Updates
  - Data Caching
  - Transaction Roll-back (Undo) and Roll-Forward
  - Checkpointing
  - Recovery schemes
    - ARIES Recovery Scheme
    - Recovery in Multidatabase System