CONTENTS

CHAPTERS	PAGE NO
1. Introduction to DBMS	3
2. ER Data Model	14
3. Relational Algebra	24
4. Relational Calculus	31
5. SQL	36
6. Relational Database Design	45
7. Query Processing Strategy	57
8. Transaction	59
9. Concurrency Control	66
10. Database Recovery Techniques	72

Introduction: What is Database Management System (DBMS)?

- Collection of interrelated data and set of programs to access or manipulate those data is known as database management system.
- The collection of data, usually referred to as the **database**, contains information relevant to an enterprise.
- The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

Database- System Applications

- Banking: For customer information, accounts, loans and all transactions.
- Airlines: For reservations, schedules information.
- Universities: For student information, course registration, and grades.
- Sales: For customers, products, purchase information.
- Manufacturing: For production, inventory, orders, supply chain.
- Human resources: For employee records, salaries, tax deductions.

Drawbacks of using file systems to store data (or Advantages of DBMS)

(i) Data redundancy and inconsistency

Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies

of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

(ii) Difficulty in accessing data

Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of all customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory. The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

(iii)Data isolation

Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

(iv)Integrity problems

The data values stored in the database must satisfy certain types of **consistency constraints**. For example, the balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new

constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

(v) Concurrent access by multiple users

For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account A, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account A at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain either $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

(vi)Atomicity Problems

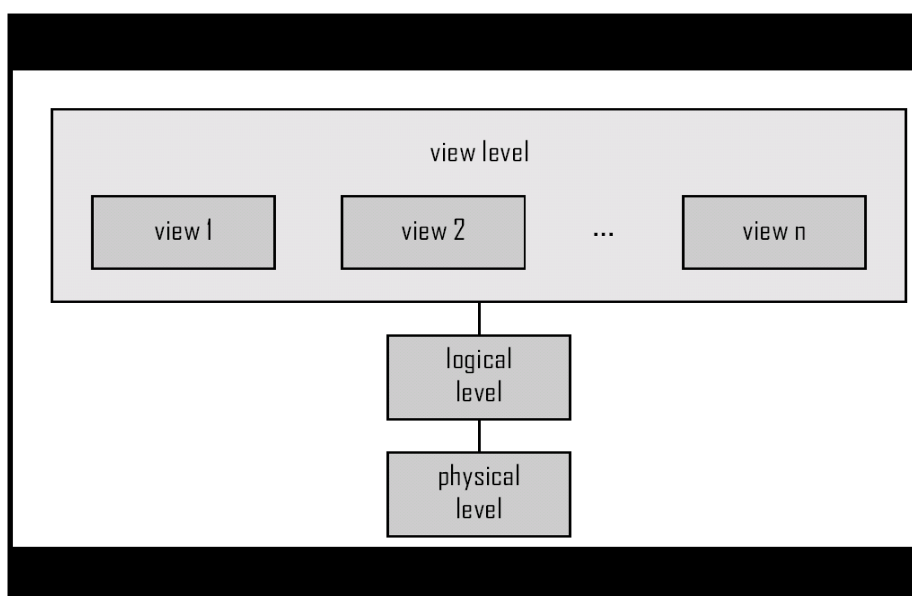
A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account A but was not credited to account B, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be atomic—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

(vii) Security problems

Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult.

Levels of Abstraction

- **Physical level:** The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.
- **Logical level:** The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures.
- **View level:** It is the highest level of data abstraction which describes only part of the entire database. The view level of abstraction exists to simplify their interaction with the system.



Instances and Schemas

- **Schema** – Schema refers to the overall design of a database.

  Example: The database consists of information about a set of customers and accounts and the relationship between them). It is analogous to type information of a variable in a program.

  - **Physical schema**: database design at the physical level
  - **Logical schema**: database design at the logical level

- **Instance** – Instance refers to the actual content of the database at a particular instance. It is analogous to the value of a variable

- **Physical Data Independence** –Physical data independence is the ability to modify the physical schema without having alteration to the logical schemas or application programs.

- **Logical Data Independence** – Logical data independence is the ability to modify the logical schema without having alteration in physical schemas or application programs.

Database Languages

1. **Data-Manipulation language**

   A data-manipulation language (DML) is a language that enables users to access or manipulate data in a database. The types of access are:

   - Retrieval of information stored in the database.
   - Insertion of new information into the database.
   - Deletion of information from the database.
   - Modification of data stored in the database.

These are basically of two types:

- **Procedural DMLs** require a user to specify what data are needed and how to get those data.
- **Declarative DMLs** (also referred to **non procedural DMLs**) require a user to specify what data are needed without specifying how to get those data.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is known a **query language**. The most widely used query language is **SQL** which will be discussed vividly later on.

Example of SQL Command:

TABLE: STUD

nam	roll	branch	dob	cgpa
Rajeeb	09m21	MECH	08/09/1988	5.6
Gobind	09m23	MECH	09/08/1991	9.8
Radha	09ee02	EEE	09/04/1988	6.5
Sam	09cs01	CSE	14/03/1998	7.4
Aakash	09et03	ETC	23/12/1990	8.4
Sufian	09cs05	CSE	04/10/1989	9.4

Example 1: Find names of all students from database STUD whose CGPA is greater than 7.

SELECT nam **FROM** Stud **WHERE** cgpa>7	**Output:** Gobind Sam Aakash Sufian

Example 2: Find names and cgpa of all students from database STUD whose branch is CSE.

SELECT nam, cgpa **FROM** Stud **WHERE** branch=CSE	**Output:** Sam 7.4 Sufian 9.4

2. **Data Definition language(DDL)**

DDL is a special language to specify a database schema or to create a database.

```
create table account
(
        account_no      char(10),
        branch_name     char(10),
        balance         integer
)
```

Output:

account_no	branch_name	balance

Database System Architecture

- A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor components.**
- The storage manager is important because databases typically require a large amount of storage space.
- The query processor is important because it helps the database system simplify and facilitate access to data.

1. Storage Manager

A storage manager is a program module that provides the interface between the low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The

raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database. The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which provide fast access to data items that hold particular values.

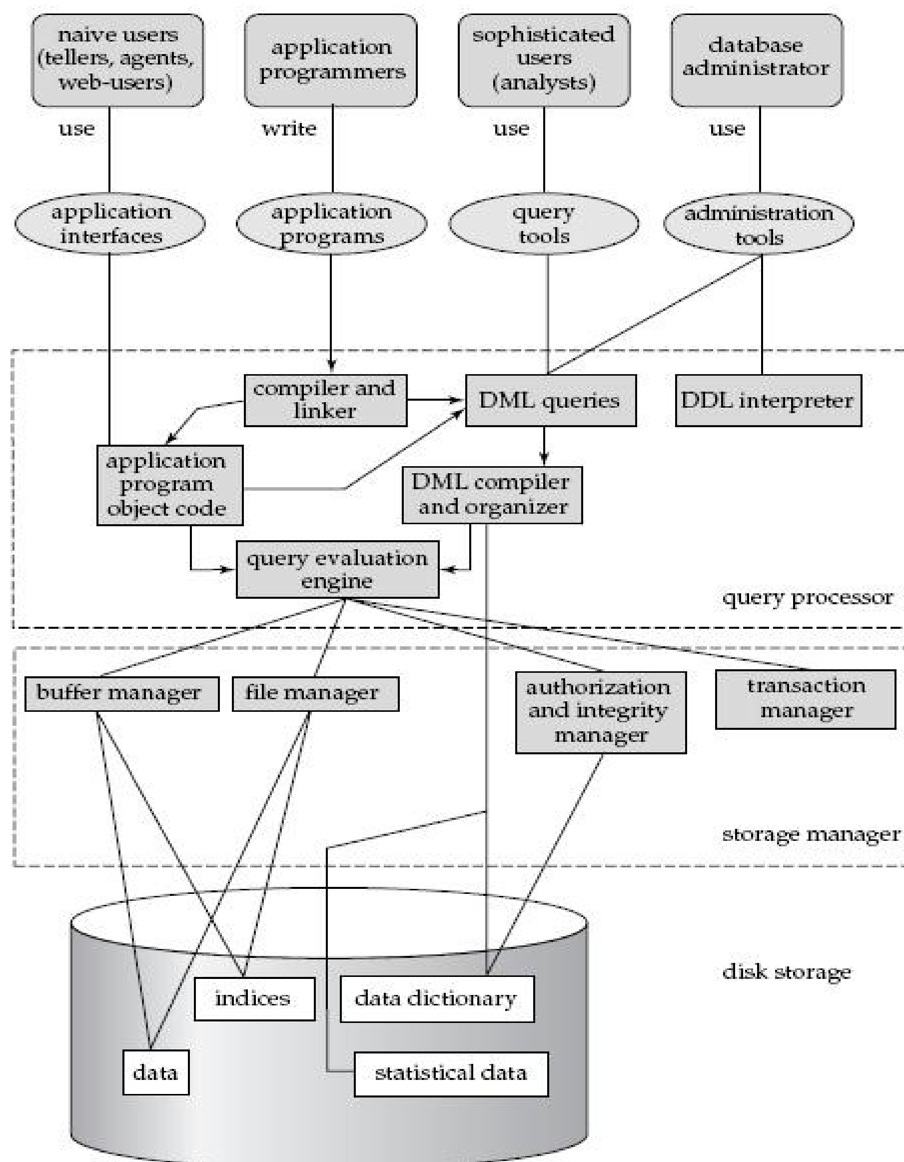
2. The Query Processor

The query processor components include

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.

- **DML** compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

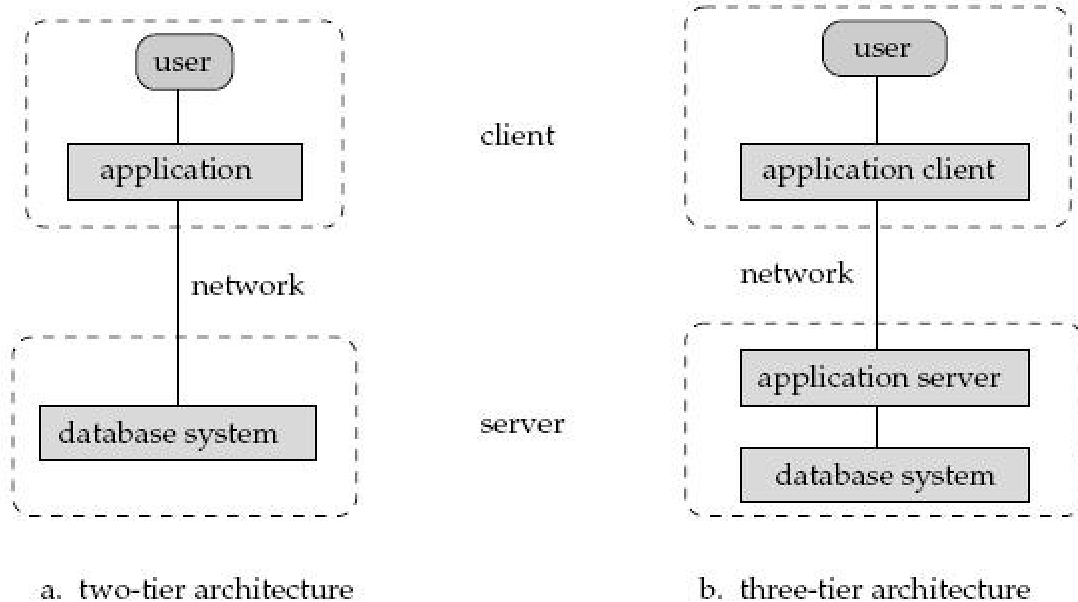
  A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.



Application Architectures

- In **two-tier architecture**, the application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

- In contrast, in **three-tier architecture**, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data.



a. two-tier architecture b. three-tier architecture

Data Models

- A collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints is known as Data Model.
- A data model provides a way to describe the design of a database at the physical, logical and view level.

Various data models are:

(i) Entity-Relationship Model

The entity–relationship model is a high-level data model. It is based on a perception of a real world that consists of a collection of basic objects, called entities, and of relationships among these objects.

(ii) Relational Data Model

- A relational database consists of a collection of tables, each of which is assigned a unique name.
- A row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name.

(iii) Object Oriented Data Model

The object-oriented model can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity.

(iv) **Network Model**

The network model is a database model conceived as a flexible way of representing objects and their relationships. Its distinguishing feature is that the schema, viewed as a graph in which object types are nodes and relationship types are arcs, is not restricted to being a hierarchy or lattice.

Entity Relationship Model

- The **entity-relationship** (**E-R**) data model perceives the real world as consisting of basic objects, called entities, and relationships among these objects.
- It was developed to facilitate database design by allowing specification of an enterprise schema, which represents the overall logical structure of a database.
- The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema.

Basic Concepts

The E-R data model employs three basic notions: entity sets, attributes and relationship sets.

(i) Entity Sets

- An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity.
- An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.
- An **entity set** is a set of entities of the same type that share the same properties, or attributes.
- The set of all persons who are customers at a given bank, for example, can be defined as the entity set customer.
- Entity sets do not need to be disjoint. For example, it is possible to define the entity set of all employees of a bank (employee) and the entity set of all customers of the bank (customer). A person entity may be an employee entity, a customer entity, both, or neither.

(ii) Attributes

- Attributes are descriptive properties possessed by each member of an entity set.

- An entity is represented by a set of **attributes**.
- Possible attributes of the customer entity set are customer-id, customer-name, customer-street, and customer-city.
- For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute.
- An attribute, as used in the E-R model, can be characterized by the following attribute types:

(i)Simple and composite attributes

Simple attributes are not divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute name could be structured as a composite attribute consisting of first-name, middle-initial, and last-name.

(ii) Single-valued and multi-valued attributes

The attributes in our examples all have a single value for a particular entity. For instance, the loan-number attribute for a specific loan entity refers to only one loan number. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Consider an employee entity set with the attribute phone-number. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be **multi-valued**.

(iii) Derived attribute

The value for this type of attribute can be derived from the values of other related attributes or entities. If the customer entity set also has an attribute date-of-birth, we can calculate age from date-of-birth and the current date. Thus, age is a derived attribute. In this case, date-of-birth may be referred to as a base attribute, or a stored attribute.

(iii) Relationship Sets

- A **relationship** is an association among several entities.
- A **relationship set** is a set of relationships of the same type. Formally, it is a mathematical relation on $n \geq 2$ (possibly non distinct) entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of $\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$ where $(e_1, e_2, \ldots, e_n)$ is a relationship.

Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. In this section, we examine **mapping cardinalities** and **participation constraints**, which are two of the most important types of constraints.

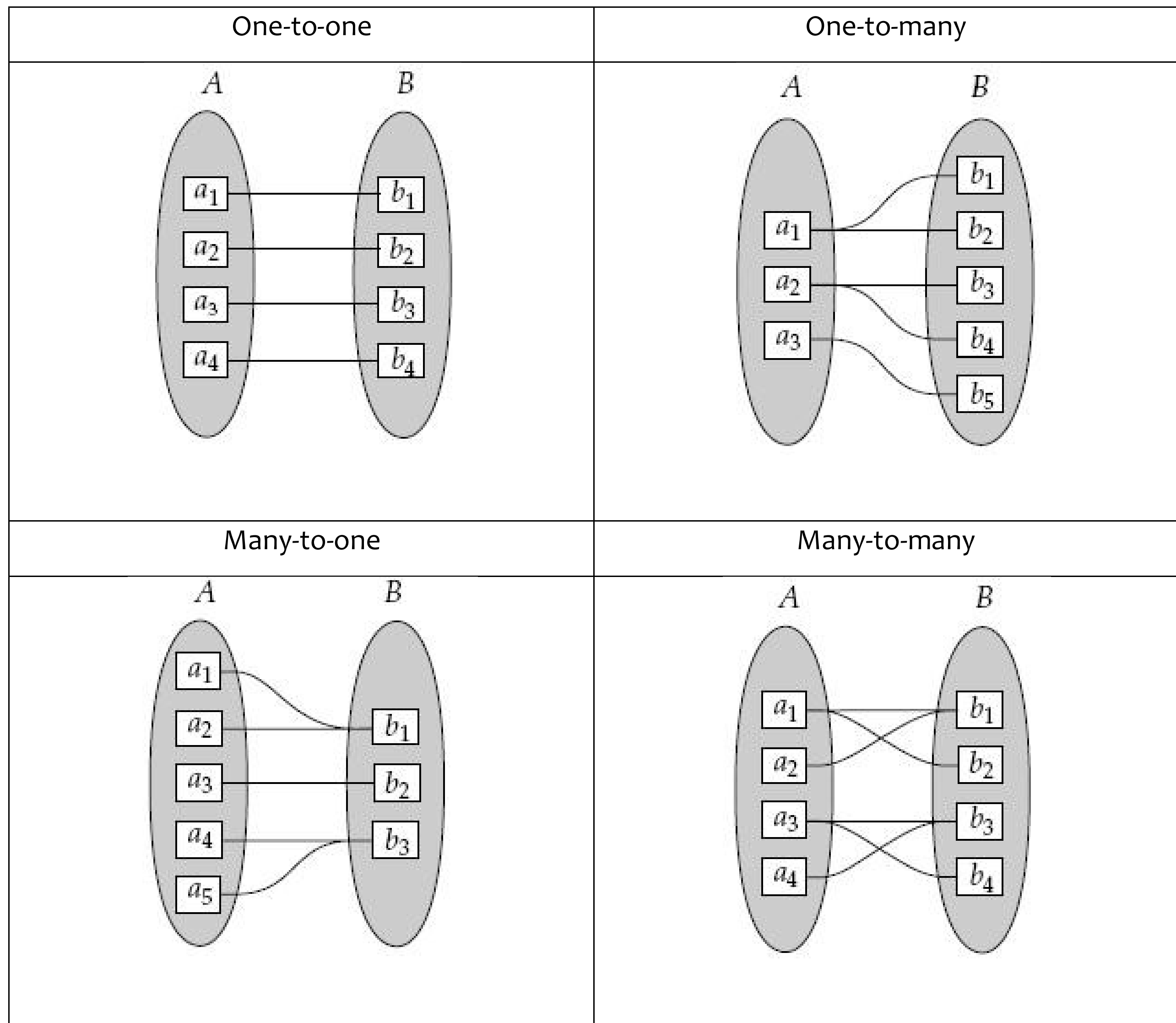
(i)Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

(a) **One to one**. An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.

(b) **One to many**. An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A.

(c) **Many to one**. An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.

(d) **Many to many**. An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



(ii) Participation Constraints

- ✓ The participation of an entity set E in a relationship set R is said to be **total** if every entity in E participates in at least one relationship in R.
- ✓ If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be **partial.**
- ✓ For example, we expect every loan entity to be related to at least one customer through the borrower relationship. Therefore the participation of loan in the relationship set borrower is total.

- ✓ In contrast, an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the customer entities are related to the loan entity set through the borrower relationship, and the participation of customer in the borrower relationship set is therefore partial.

For below examples consider following BANKING ENTERPRISE schemas:

Branch (branch-name, branch-city, assets)
Customer (customer-name, customer-street, customer-city)
Loan (loan-number, branch-name, amount)
Borrower (customer-name, loan-number)
Account (account-number, branch-name, balance)
Depositor (customer-name, account-number)

Keys

- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set.
- For example, the customer-id attribute of the entity set customer is sufficient to distinguish one customer entity from another. Thus, customer-id is a superkey. Similarly, the combination of customer-name and customer-id is a superkey for the entity set customer. The customer-name attribute of customer is not a superkey, because several people might have the same name.
- The concept of a superkey is not sufficient for our purposes, since, as we saw, a superkey may contain extraneous attributes. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

- It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of customer-name and customer-street is sufficient to distinguish among members of the customer entity set. Then, both {customer-id} and {customer-name, customer-street} are candidate keys. Although the attributes customerid and customer-name together can distinguish customer entities, their combination does not form a candidate key, since the attribute customer-id alone is a candidate key.
- **A primary key d**enote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set.

Entity Relationship(ER) Diagram

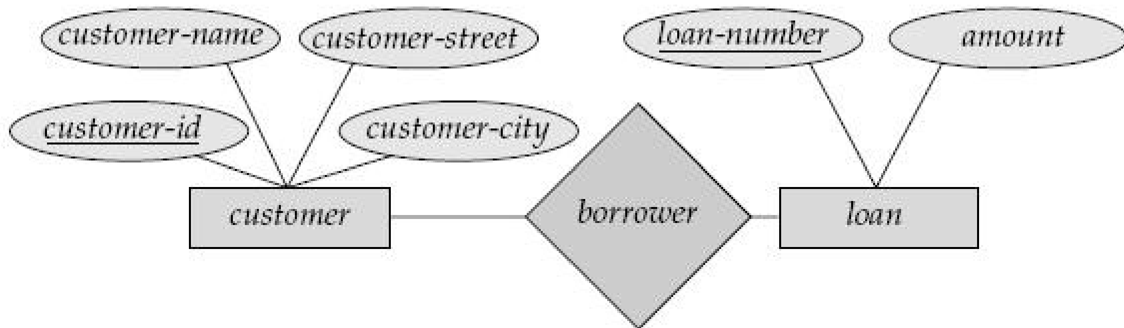
An **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model. Such a diagram consists of the following major components:

- ✓ **Rectangles**, which represent entity sets
- ✓ **Ellipses**, which represent attributes
- ✓ **Diamonds**, which represent relationship sets
- ✓ **Lines**, which link attributes to entity sets and entity sets to relationship sets
- ✓ **Double ellipses**, which represent multivalued attributes
- ✓ **Dashed ellipses**, which denote derived attributes
- ✓ **Double lines**, which indicate total participation of an entity in a relationship set
- ✓ **Double rectangles**, which represent weak entity sets

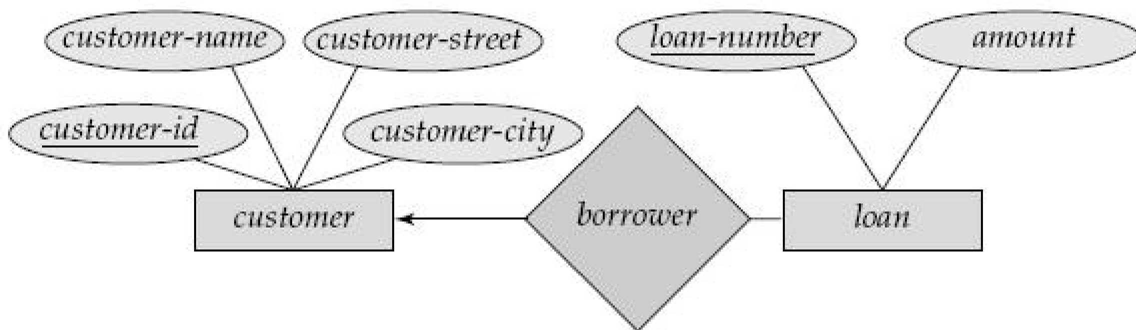
Consider the entity-relationship diagram, which consists of two entity sets, **customer** and **loan**, related through a binary relationship set borrower. The attributes associated with customer are customer-id, customer-name, customer-street, and customer-city. The attributes associated with loan are loan-number and amount. In ER diagram, attributes of an

entity set that are members of the **primary key are underlined**. The relationship set borrower may be many-to-many, one-to-many, many-to-one, or one-to-one.

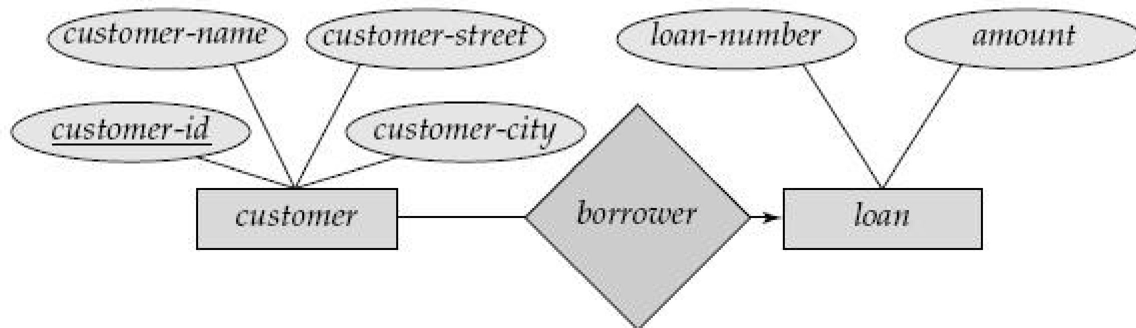
One-to-one



One-to-many



Many-to-one



Many-to-many

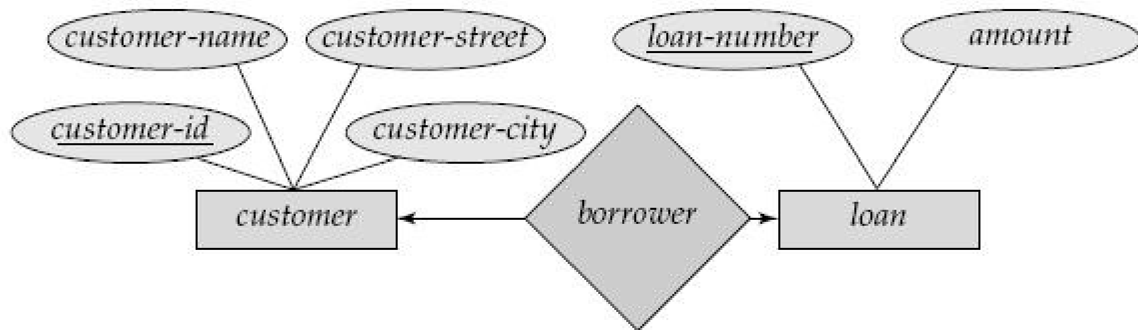


Figure1 shows how composite attributes can be represented in the E-R notation. Here, a **composite attribute** name, with component attributes first-name, middle-initial, and last-name replaces the simple attribute customer-name of customer. Also, a composite attribute address, whose component attributes are street, city, state, and zip-code replaces the attributes customer-street and customer-city of customer. The attribute street is itself a composite attribute whose component attributes are street-number, street-name, and apartment number. Figure 1 also illustrates a **multi-valued attribute** phone-number, depicted by a double ellipse, and a **derived attribute** age, depicted by a dashed ellipse.

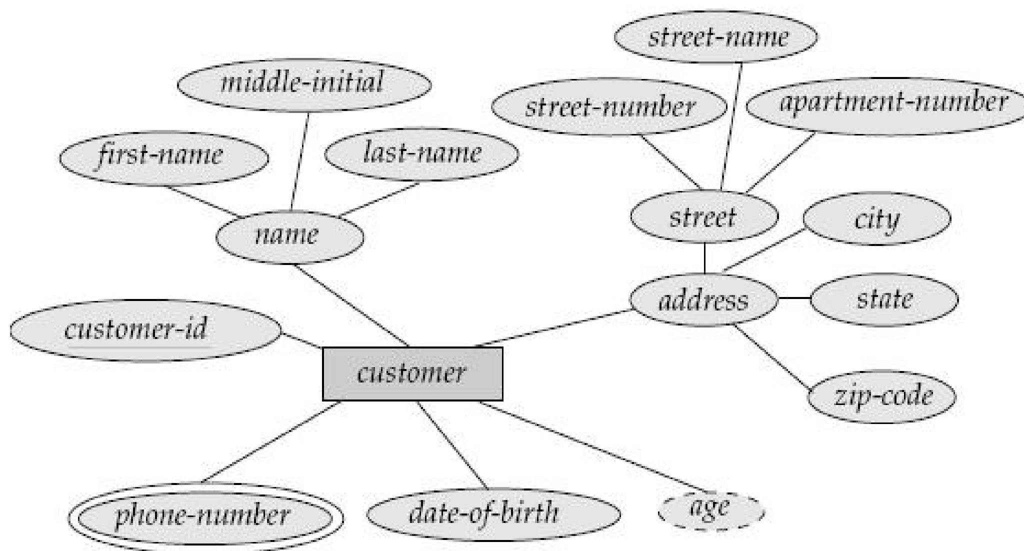
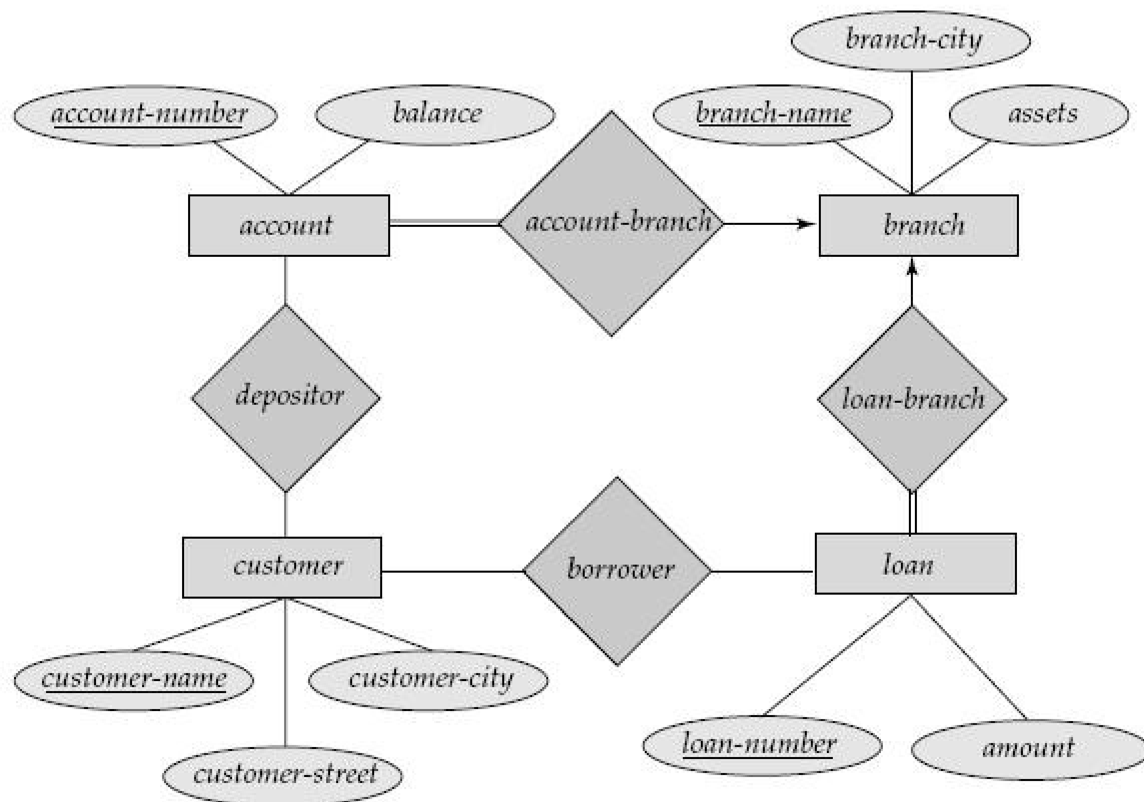


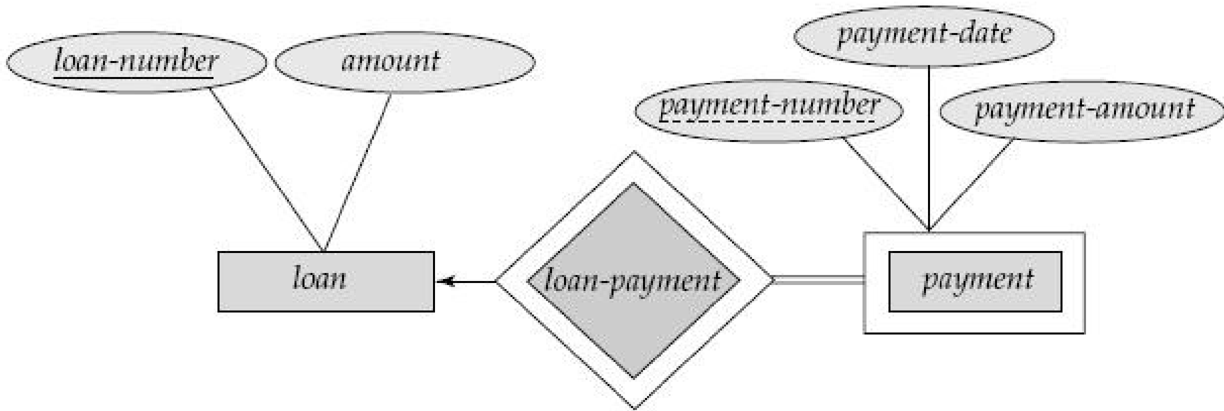
Figure 1

E-R diagram for the banking enterprise



Weak and Strong Entity Set

- ✓ An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.
- ✓ As an illustration, consider the entity set payment, which has the three attributes: **payment-number, payment-date,** and **payment-amoun**t. Payment numbers are typically sequential numbers, starting from 1, generated separately for each loan. Thus, although each payment entity is distinct, payments for different loans may share the same payment number. Thus, this entity set does not have a primary key; it is a weak entity set.


loan-number
amount
payment-date
payment-number
payment-amount
loan
loan-payment
payment

Relational Algebra

- The relational algebra is a procedural query language.
- It consists of a set of operations that take one or two relations (i.e. tables) as input and produce a new relation (i.e. table) as their result.
- The fundamental operations in the relational algebra are **select, project, union, set difference, Cartesian product, and rename.**
- In addition to the fundamental operations, there are several other operations—namely, **set intersection, join, division, and assignment**.

Fundamental Operation

- The **select, project,** and **rename** operations are called **unary** operations, because they operate on one relation (or table).
- **Union, set difference** and **Cartesian product** operations operate on pairs of relations (or tables) and are, therefore, called **binary** operations.

For all the below examples use following database schemas:

Branch (branch-name, branch-city, assets)

Customer (cust-name, cust-street, cust-city)

Loan (loan-number, branch-name, amount)

Borrower (cust-name, loan-number)

Account (account-number, branch-name, balance)

Depositor (cust-name, account-number)

branch

branch-name	branch-city	assets
Proof Road	Balasore	6000
Evening	Balasore	7000
Charampa	Bhadrak	5000
Main	Balasore	12000

customer

cust-name	cust-street	cust-city
Shaswati	DRDO Colony	Balasore
Nivedita	Remuna	Balasore
Veena	Bant Chhaka	Bhadrak
Heera	Baramunda	BBSR

loan

loan-number	branch-name	amount
L 01	Proof Road	2000
L 02	Proof Road	2500
L 03	Evening	4500

borrower

cust-name	loan-number
Shaswati	L 02
Nivedita	L 01
Veena	L 03

account

account-number	branch-name	balance
A 01	Evening	900
A 02	Proof Road	1100
A 03	Main	500

depositer

cust-name	account-number
Shaswati	A 03
Nivedita	A 01
Heera	A 02

1. Select Operation(σ)

- The **select** operation selects tuples /rows that satisfy a given predicate (condition).
- We use the lowercase Greek letter sigma (σ) to denote selection.
- The predicate appears as a subscript to σ i.e. σ_p where p is the predicate/ condition.
- The argument relation is in parentheses after the σ.
- In general, we allow comparisons using =, _=, <, ≤, >, ≥ in the selection predicate.
- Furthermore, we can combine several predicates into a larger predicate by using the connectives and (^), or (v), and not (¬).

Example 1: Select all tuples (or rows) of the loan relation where the branch is "Proof Road".

Solution: $\sigma_{\text{branch-name} = \text{"Proof Road"}}$ (loan)

loan-number	branch-name	amount
L 01	Proof Road	2000
L 02	Proof Road	2500

Example 2: Find all tuples of loan relation in which the loan amount is more than 2000.

Solution: $\sigma_{\text{amount} > 200\ 0}$ (loan)

loan-number	branch-name	amount
L 02	Proof Ro d	2500
L 03	Evening	4500

Example 3: Find all tuples pertaining to loans of more than 2000 made by the Proof Road branch.

Solution: $\sigma_{\text{branch-name} = \text{"Proof Road"} \wedge \text{amount} > 2000}$ (loan)

loan-number	branch-name	amount
L 02	Proof Road	2500

2. Project Operation (Π)

- The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated.
- Projection is denoted by the uppercase Greek letter pi (Π).

- We list those attributes that we wish to appear in the result as a subscript to Π.

Example 4: List all loan numbers and the amount of the loan relation.

Solution: $\Pi_{\text{loan-number, amount}}(\text{loan})$

loan-number	amount
L 01	2000
L 02	2500
L 03	4500

Example 5: Find name of those customers who live in Remuna.

Solution: $\Pi_{\text{cust-name}}(\sigma_{\text{cust-city} = \text{"Remuna"}}(\text{customer}))$

cust-name
Nivedita

3. Union(U)

The union of two relation r and s, r U s is valid if:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the ith attribute of r and the ith attribute of s must be the same, for all i.

Example 6: Find the names of all bank customers who have either an account or a loan or both.

Solution: $\Pi_{\text{cust-name}}(\text{borrower}) \cup \Pi_{\text{cust-name}}(\text{depositor})$

4. Set difference(---)

- The set-difference operation, denoted by −, allows us to find tuples that are in one relation but are not in another.
- The expression r − s produces a relation containing those tuples in r but not in s.

Example 6: Find all customers of the bank who have an account but not a loan.

Solution: $\Pi_{\text{cust-name}}$ (depositor) − $\Pi_{\text{cust-name}}$ (borrower)

cust-name
Heera

5. Cartesian Product(×)

- The **Cartesian-product** operation, denoted by a cross (×), allows us to combine information from two relations (tables).
- We write the Cartesian product of relations r1 and r2 as r1 × r2.
- If r1 has m attributes and r2 has n attributes, then r1 × r2 has m+n attributes.
- If r1 has p tuples and r2 has q tuples, then r1 × r2 has p × q tuples.

For example, the relation schema for r = **borrower × loan** is (borrower.cust-name, borrower.loan-number, loan.loan-number, loan.branch-name, loan.amount).

With this schema, we can distinguish borrower.loan-number from loan.loan-number. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as (cust-name, borrower.loan-number, loan.loan-number, branch-name, amount).

cust-name	borrower.loan-number	loan.loan-number	branch-name	amount
Shaswati	L 02	L 01	Proof Road	2000
Shaswati	**L 02**	**L 02**	**Proof Road**	**2500**
Shaswati	L 02	L 03	Evening	4500
Nivedita	**L 01**	**L 01**	**Proof Road**	**2000**
Nivedita	L 01	L 02	Proof Road	2500
Nivedita	L 01	L 03	Evening	4500
Veena	L 03	L 01	Proof Road	2000
Veena	L 03	L 02	Proof Road	2500
Veena	**L 03**	**L 03**	**Evening**	**4500**

Example 7: Find name of customers who have a loan at the Proof Road branch.

Solution:

$\Pi_{\text{cust-name}}(\sigma_{\text{borrower.loan-number=loan.loan-number}}(\sigma_{\text{branch-name="Proof Road"}}(\text{borrower} \times \text{loan})))$

6. Rename(ρ)

- Given a relational-algebra expression E, the expression ρ_x (E) returns the result of expression E under the name x.
- A second form of the rename operation is as follows. Assume that a relational algebra expression E has arity n. Then, the expression $\rho_x(A1,A2,...,An)$ (E) returns the result of expression E under the name x, and with the attributes renamed to A1,A2, . . .,An.

Example 8: Find the largest account balance in the bank.

Solution:

$\Pi_{\text{balance}}(\text{account}) - \Pi_{\text{account .balance}}(\sigma_{\text{account .balance < d.balance}}(\text{account} \times \rho_d(\text{account})))$

account.account-number	account.branch-name	account.balance	d. account-number	d. branch-name	d. balance
A 01	Evening	900	A 01	Evening	900
A 01	**Evening**	**900**	**A 02**	**Proof Road**	**1100**
A 01	Evening	900	A 03	Main	500
A 02	Proof Road	1100	A 01	Evening	900
A 02	Proof Road	1100	A 02	Proof Road	1100
A 02	Proof Road	1100	A 03	Main	500
A 03	**Main**	**500**	**A 01**	**Evening**	**900**
A 03	**Main**	**500**	**A 02**	**Proof Road**	**1100**
A 03	Main	500	A 03	Main	500

{1100, 900, 500} - {900, 500}=1100 (Answer)

Relational Calculus

Tuple relational Calculus:

A query in the tuple relational calculus is expressed as:

$\{t \mid P(t)\}$ that is, it is the set of all tuples t such that predicate P is true for t.

Example 1: Write a query in relational calculus to find the branch-name, loan-number, and amount for loans of over 1200.

Solution:

$\{t \mid t \in loan \land t[amount] > 1200\}$

Suppose that we want only the loan-number attribute, rather than all attributes of the loan relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (loan-number). We need those tuples on (loan-number) such that there is a tuple in loan with the amount attribute > 1200. To express this request, we need the construct "there exists" from mathematical logic.

The notation $\exists\ t \in r\ (Q(t))$ means "there exists a tuple t in relation r such that predicate Q(t) is true."

Example 2: Write a query in relational calculus to find loan-number for loans of over 1200.

Solution:

$\{ t \mid \exists s \in loan\ (t[loan\text{-}number] = s[loan\text{-}number] \land s[amount] > 1200) \}$

Example 3: Write a query in relational calculus to find the names of all customers who have a loan from the Balasore branch.

Solution:

This query is slightly more complex than the previous queries, since it involves two relations:

borrower and loan. As we shall see, however, all it requires is that we have two "there exists" clauses in our tuple-relational-calculus expression, connected by and (∧).We write the query as follows:

{ t | ∃s ∈ loan (t[loan-number] = s[loan-number] ∧ s[amount] > 1200) }

Example 4: Write a query in relational calculus to to find all customers who have a loan, an account, or both at the bank.

Solution:

{t | ∃ s ∈ borrower (t[customer-name] = s[customer-name])

∨ ∃ u ∈ depositor (t[customer-name] = u[customer-name])}

Example 5: Write a query in relational calculus to find all those customers who have both an account and a loan at the bank.

Solution:

{ t | ∃ s ∈ borrower (t[customer-name] = s[customer-name])

∧∃ u ∈ depositor (t[customer-name] = u[customer-name]) }

Example 6: Write a query in relational calculus to Find all customers who have an account at the bank but do not have a loan from the bank.

Solution:

{t | ∃ u ∈ depositor (t[customer-name] = u[customer-name])

∧ ¬ ∃ s ∈ borrower (t[customer-name] = s[customer-name])}

Formal Definition: A tuple-relational-calculus expression is of the form {t | P(t)}, where P is a formula. Several tuple variables may appear in a formula. A tuple variable is said to be a free variable unless it is quantified by a ∃ or ∀. Thus, in t ∈ loan ∧ ∃s ∈ customer(t[branch-name] = s[branch-name]) and t is a free variable. Tuple variable s is said to be a bound variable.

A tuple-relational-calculus formula is built up out of atoms. An atom has one of the following forms:

- s ∈ r, where s is a tuple variable and r is a relation (we do not allow use of the ∈ operator)
- s[x] Θ u[y], where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator (<, ≤, =, _=, >, ≥); we require that attributes x and y have domains whose members can be compared by Θ
- s[x] Θ c, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P1 is a formula, then so are ¬ P1 and (P1).
- If P1 and P2 are formulae, then so are P1 ∨ P2, P1 ∧ P2, and P1 ⇒ P2.
- If P1(s) is a formula containing a free tuple variable s, and r is a relation, then ∃ s ∈ r (P1(s)) and ∀ s ∈ r (P1(s)) are also formulae.

Example 7: Let the following relation schemas be given:

R = (A,B,C)

S = (D,E, F)

Let relations r(R) and s(S) be given. Give an expression in the tuple relational calculus that is equivalent to each of the following:

a. $\Pi_A(r)$

b. $\sigma_{B=17}(r)$

c. $r \times s$

d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

Solution:

a. $\{t \mid \exists q \in r\ (q[A] = t[A])\}$

b. $\{t \mid t \in r \wedge t[B] = 17\}$

c. $\{t \mid \exists p \in r\ \exists q \in s\ (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = q[D]$
$\wedge t[E] = q[E] \wedge t[F] = q[F])\}$

d. $\{t \mid \exists p \in r\ \exists q \in s\ (t[A] = p[A] \wedge t[F] = q[F] \wedge p[C] = q[D]\}$

The Domain Relational Calculus

A second form of relational calculus, called **domain relational calculus**, uses domain variables that take on values from an attributes domain, rather than values for an entire tuple.

An expression in the domain relational calculus is of the form

$\{< x1, x2, \ldots, xn > \mid P(x1, x2, \ldots, xn)\}$

where x1, x2, . . . , xn represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- $< x1, x2, \ldots, xn > \in r$, where r is a relation on n attributes and x1, x2, . . . , xn are domain variables or domain constants.
- $x \Theta y$, where x and y are domain variables and Θ is a comparison operator (<, ≤, =, _=, >, ≥). We require that attributes x and y have domains that can be compared by Θ.
- $x \Theta c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

- An atom is a formula.

- If P1 is a formula, then so are $\neg$ P1 and (P1).
- If P1 and P2 are formulae, then so are P1 $\vee$ P2, P1 $\wedge$ P2, and P1 $\Rightarrow$ P2.
- If P1(x) is a formula in x, where x is a domain variable, then $\exists$ x (P1(x)) and $\forall$ x (P1(x)) are also formulae.

Example 1: Write a query in domain relational calculus to find the branch-name, loan-number, and amount for loans of over $1200.

Solution:

$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$

Example 2: Write a query in domain relational calculus to find loan-number for loans of over $1200.

Solution:

$\{< l > \mid \exists a (< l, b, a > \in loan \wedge a > 1200)\}$

Example 3: Write a query in domain relational calculus to find the names of all customers who have a loan from the Proof Road branch. Also find the loan amount.

Solution:

$\{< c,a > \mid \exists l (< c, l > \in borrower \wedge \exists b (< l, b, a > \in loan \wedge b = \text{"Proof Road"}))\}$

Example 4: Write a query in domain relational calculus to to find all customers who have a loan, an account, or both at proof road branch.

Solution:

$\{< c > \mid \exists l (< c, l > \in borrower$

$\wedge \exists b, a (< l, b, a > \in loan \wedge b = \text{"Proof Road"}))$

$\vee \exists a (< c,a > \in depositor$

$\wedge \exists b, n (< a, b, n > \in account \wedge b = \text{"Proof Road"}))\}$

SQL

The basic structure of an SQL expression consists of three clauses: **select**, **from**, and **where**.

- The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- The **from** clause corresponds to the Cartesian-product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

A typical SQL query has the form:

select A1, A2, . . .,An

from r1, r2, . . . , rm

where P

Each Ai represents an attribute, and each ri a relation. P is a predicate.

The above query is equivalent to the relational-algebra expression:

$\Pi_{A1, A2,...,An}(\sigma_P (r1 \times r2 \times \cdot \cdot \cdot \times rm))$

The enterprise that we use in the examples in this chapter, and later chapters, is a banking enterprise with the following relation schemas:

Branch-schema = (branch-name, branch-city, assets)

Customer-schema = (customer-name, customer-street, customer-city)

Loan-schema = (loan-number, branch-name, amount)

Borrower-schema = (customer-name, loan-number)

Account-schema = (account-number, branch-name, balance)

Depositor-schema = (customer-name, account-number)

The Select Clause

- The asterisk symbol * can be used to denote "all attributes."
- The **select** clause may also contain arithmetic expressions involving the operators +, −, *, and / operating on constants or attributes of tuples.
- If you want to force the elimination of duplicates, insert the keyword **distinct** after **select.**

Example 1: Find the names of all branches in the loan relation.

Solution:

select branch-name

from loan

The Where Clause

- SQL uses the logical connectives **and**, **or**, and **not**—rather than the mathematical symbols in the **where** clause.
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.

Example 2: Find the loan number of those loans with loan amounts between 90,000 and 100,000.

Solution:

select loan-number

from loan

where amount **between** 90000 **and** 100000

Alternative way

select loan-number

from loan

where amount <= 100000 **and** amount >= 90000

The from Clause

- The **from** clause by itself defines a Cartesian product of the relations in the clause.
- It lists the relations to be scanned in the evaluation of the expression.

Example 3: Find all customers who have a loan from the bank, find their names, loan numbers and loan amount.

Solution:

select customer-name, borrower.loan-number, amount

from borrower, loan

where borrower.loan-number = loan.loan-number

Example 4: Find the customer names, loan numbers, and loan amounts for all loans at the Proof Road branch.

Solution:

select customer-name, borrower.loan-number, amount

from borrower, loan

where borrower.loan-number = loan.loan-number **and** branch-name = ' Proof Road'

The Rename Operation

- SQL provides a mechanism for renaming both relations and attributes. It uses the **as** clause, taking the form:

  old-name as new-name
- The **as** clause can appear in both the **select** and **from** clauses.

For example, if we want the attribute name loan-number to be replaced with the name loan-id, we can rewrite the preceding query as:

```
select customer-name, borrower.loan-number as loan-id, amount
from borrower, loan
where borrower.loan-number = loan.loan-number
```

String Operations

The most commonly used operation on strings is pattern matching using the operator **like**. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore ( ): The character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Perry%' matches any string beginning with "Perry".
- '%idge%' matches any string containing "idge" as a substring, for example, Perryridge', 'Rock Ridge', 'Mianus Bridge', and 'Ridgeway'.
- '- - -' matches any string of exactly three characters.
- '- - -%'matches any string of at least three characters.

Example 5: Find the names of all customers whose street address includes the substring 'Main'.

Solution:

select customer-name

from customer

where customer-street **like** '%Main%'

Ordering the Display of Tuples

- SQL offers the user some control over the order in which tuples in a relation are displayed.
- The **order by** clause causes the tuples in the result of a query to appear in sorted order.

Example 6: List in alphabetic order all customers who have a loan at the Perryridge branch.

Solution:

select distinct customer-name

from borrower, loan

where borrower.loan-number = loan.loan-number **and** branch-name = 'Perryridge'

order by customer-name

Example 7: List the entire loan relation in descending order of amount.

Solution:

select *

from loan

order by amount **desc**, loan-number **asc**

Set Operations

- The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the relational-algebra operations ∪, ∩, and −.
- Like union, intersection, and set difference in relational algebra, the relations participating in the operations must be compatible; that is, they must have the same set of attributes.

The Union Operation

- The **union** operation automatically eliminates duplicates, unlike the **select** clause.

Example 8: find all customers having a loan, an account, or both at the bank.

Solution:

(**select** customer-name
from depositor)
union
(**select** customer-name
from borrower)

If we want to retain all duplicates, we must write **union all** in place of **union**:

(**select** customer-name
from depositor)
union all
(**select** customer-name
from borrower)

The Intersect Operation

- The **intersect** operation automatically eliminates duplicates

Example 9: find all customers who have both a loan and an account at the bank.

Solution:

(**select distinct** customer-name

from depositor)

intersect

(**select distinct** customer-name

from borrower)

If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

(**select** customer-name

from depositor)

intersect all

(**select** customer-name

from borrower)

The Except Operation

The **except** operation automatically eliminates duplicates.

Example 10: Find all customers who have an account but no loan at the bank.

Solution:

(**select distinct** customer-name

from depositor)

except

(**select** customer-name

from borrower)

If we want to retain all duplicates, we must write **except all** in place of **except:**

(**select** customer-name

from depositor)

except all

(**select** customer-name

from borrower)

Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

Average: **avg**

Minimum: **min**

Maximum: **max**

Total: **sum**

Count: **count**

Example 11: Find the average account balance at the Proof Road branch.

Solution:

select avg (balance)

from account

where branch-name = 'Proof Road'

Example 12: Find the average account balance at each branch.

Solution:

select branch-name, **avg** (balance)

from account

group by branch-name

Example 13: Find the number of depositors for each branch.

Solution:

select branch-name, **count** (**distinct** customer-name)

from depositor, account

where depositor.account-number = account.account-number

group by branch-name

Relational Database Design

- In general, the goal of a relational-database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy (repetition), yet also allows us to retrieve information easily.
- One approach is to design schemas that are in an appropriate normal form. To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. In this chapter, we introduce the notion of functional dependencies. We then define normal forms in terms of functional dependencies and other types of data dependencies.

Question 1: Give three goals of relational-database design.

Answer: The three design goals of relational databases are to avoid

- Repetition of information
- Inability to represent information
- Loss of information.

Before we continue our discussion of normal forms, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

Question 2: Explain what is meant by repetition of information and inability to represent information. Explain why each of these properties may indicate a bad relational database design.

Answer:

- Repetition of information is a condition in a relational database where the values of one

attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation. This is a bad relational database design because it increases the storage required for the relation and it makes updating the relation more difficult.

• Inability to represent information is a condition where a relationship exists among only a proper subset of the attributes in a relation. This is bad relational database design because all the unrelated attributes must be filled with null values otherwise a tuple without the unrelated information cannot be inserted into the relation.

• Loss of information is a condition of a relational database which results from the decomposition of one relation into two relations and which cannot be combined to recreate the original relation. It is a bad relational database design because certain queries cannot be answered using the reconstructed relation that could have been answered using the original relation.

Functional Dependency

- Functional dependencies play a key role in differentiating good database designs from bad database designs.
- A **functional dependency** is a type of constraint that is a generalization of the notion of key.

We defined the notion of a superkey as follows. Let R be a relation schema. A subset K of R is a **superkey** of R if, in any legal relation rI, for all pairs t1 and t2 of tuples in r such that $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal relation rI may have the same value on attribute set K.

The notion of functional dependency generalizes the notion of super key.

Definition: Consider a relation schema R, and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **functional dependency** $\alpha \rightarrow \beta$ holds on schema R if, in any legal relation rI, for all pairs of tuples t1 and t2 in r such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Let us consider the below relation r:

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d3
a2	b2	c2	d4
a3	b3	c2	d5

(Relation r)

Observe that A → C is satisfied. There are two tuples that have an A value of a1. These tuples have the same C value—namely, c1. Similarly, the two tuples with an A value of a2 have the same C value, c2. There are no other pairs of distinct tuples that have the same A value.

The functional dependency C → A is not satisfied, however. To see that it is not, consider the tuples t1 = (a2, b3, c2, d3) and t2 = (a3, b3, c2, d4). These two tuples have the same C values, c2, but they have different A values, a2 and a3, respectively. Thus, we have found a pair of tuples t1 and t2 such that t1[C] = t2[C], but t1 [A] ≠t2 [A].

Some functional dependencies are said to be **trivial** because they are satisfied by all relations. For example, A → A is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally, we see that, for all tuples t1 and t2 such that t1[A] = t2[A], it is the case that t1[A] = t2[A]. Similarly, AB → A is satisfied by all relations involving attribute A. In general, a functional dependency of the form $\alpha \rightarrow \beta$ is **trivial** if $\beta \subseteq \alpha$.

In general, a functional dependency of the form $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

Questions: Why certain functional dependencies are called trivial functional dependencies?

Answer: Certain functional dependencies are called trivial functional dependencies because they are satisfied by all relations.

In the banking example, our list of **dependencies** includes the following:

On Branch-schema:

branch-name →branch-city

branch-name →assets

On Customer-schema:

customer-name→ customer-city

customer-name→ customer-street

On Loan-schema:

loan-number → amount

loan-number → branch-name

On Borrower-schema:

No functional dependencies

On Account-schema:

account-number → branch-name

account-number → balance

On Depositor-schema:

No functional dependencies

Armstrong' s Axioms

We can use the following three rules to find logically implied functional dependencies. By applying these rules repeatedly, we can find all of F+, given F. This collection of rules is called **Armstrong's axioms** in honor of the person who first proposed it.

Reflexivity rule. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.

Augmentation rule. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.

Transitivity rule. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Proof of Armstrong's Axioms

The definition of functional dependency is: $\alpha \rightarrow \beta$ holds on R if in any legal relation r(R), for all pairs of tuples t1 and t2 in r such that t1 [α] = t2 [α], it is also the case that t1[β] = t2[β].

Reflexivity rule: if α is a set of attributes, and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.

Assume ∃t1 and t2 such that t1[α] = t2[α]

t1[β] = t2[β] since $\beta \subseteq \alpha$

$\alpha \rightarrow \beta$ definition of FD

Augmentation rule: if $\alpha \rightarrow \beta$, and γ is a set of attributes, then $\gamma \alpha \rightarrow \gamma \beta$.

Assume ∃t1, t2 such that t1[γ α] = t2[γ α]

t1[γ] = t2[γ] $\gamma \subseteq \gamma \alpha$

t1[α] = t2[α] $\alpha \subseteq \gamma \alpha$

t1[β] = t2[β] definition of $\alpha \rightarrow \beta$

t1[γ β] = t2[γ β] $\gamma \beta = \gamma \cup \beta$

$\gamma \alpha \rightarrow \gamma \beta$ definition of FD

Transitivity rule: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.

Assume ∃t1, t2 such that t1[α] = t2[α]

t1[β] = t2[β] definition of $\alpha \rightarrow \beta$

t1[γ] = t2[γ] definition of $\beta \rightarrow \gamma$

$\alpha \rightarrow \gamma$ definition of FD

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of *F+*. To simplify matters further, we list additional rules.

Union rule. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.

Decomposition rule. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.

Pseudo transitivity rule. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Proof:

Union rule.

$\alpha \rightarrow \beta$ given

$\alpha\alpha \rightarrow \alpha\beta$ augmentation rule

$\alpha \rightarrow \alpha\beta$ union of identical sets

$\alpha \rightarrow \gamma$ is given

$\alpha\beta \rightarrow \gamma\beta$ augmentation rule

$\alpha \rightarrow \beta\gamma$ transitivity rule

Decomposition rule:

$\alpha \rightarrow \beta\gamma$ given

$\beta\gamma \rightarrow \beta$ reflexivity rule

$\alpha \rightarrow \beta$ transitivity rule

$\beta\gamma \rightarrow \gamma$ reflexive rule

$\alpha \rightarrow \gamma$ transitive rule

Pseudo transitivity rule:

$\alpha \rightarrow \beta$ given

$\alpha\gamma \rightarrow \gamma\beta$ augmentation rule and set union commutativity

$\gamma\beta \rightarrow \delta$ given

$\alpha\gamma \rightarrow \delta$ transitivity rule

Closure of a Set of Functional Dependencies

Let F be a set of functional dependencies. The **closure** of F, denoted by F+, is the set of all functional dependencies logically implied by F.

Question 3: You are given a relation schema R = (A, B, C, D, E) and the set of functional dependencies F :

A →BC

CD →E

B→D

E→ A

Find closure of F (F^+).

Solution:

E→ A and A →BC, using transitivity rule we have

E→BC

CD→ E and E→A, using transitivity rule we have

CD→A

A →BC, using decomposition rule we have

A→ B and A →C

B→D and CD →E, using pseudo-transitivity rule we have

BC→E

Therefore, F^+={ E→BC, CD→A, A→ B, A →C, BC→E}

Closure of Attribute Sets

Let R be a relation with set of functional dependency F. Let α be a set of attributes. The closure of attribute set α under a set of FD is denoted by α^+.

How to find α^+

1. Set Result= α
2. Repeat until Result does not change

   For all FDs X→Y, if X⊆ Result

   Result=Result U Y

Question 4: Suppose you are given a relation schema R = (A, B, C, G, H, I) and the set of functional dependencies:

A →B

A →C

CG→ H

CG→ I

$B \rightarrow H$

Find out closure of A^+, CG^+ and B^+.

Solution:

A^+=ABCH

CG^+=CGHI

B^+=BH

Prime attribute and non-prime attribute

An attribute in a relation schema R is a **prime** attribute if A is a part of any candidate key of the relation.

An attribute in a relation schema R is a **non-prime** attribute if A is not a part of any candidate key of the relation.

Question 5: Given R (A, B, C, D, E, H) and F= {A→BC, CD→E, E→C, AH→D}**. Find candidate key, prime attributes and non-prime attributes of R.**

Solution:

A^+ =ABC

CD^+=CDE

E^+=EC

AH^+=ABCDEH=R

Hence AH is candidate key.

The attributes A and H are prime attributes and the attributes B, C, D, E are non prime attribute.

1st Normal Form (1 NF): [Remove multivalued attributes]

- A relation schema R is in **first normal form** (1NF) if the domains of all attributes of R are atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units.
- A set of names is an example of a non atomic value. For example, if the schema of a relation employee included attribute children whose domain elements are sets of names, the schema would not be in first normal form.
- Composite attributes, such as an attribute address with component attributes street and city, also have non atomic domains.

[TABLE: Student]

Name	Roll	Branch	Subject
Jayashree	001	CSE	Java, C, DBMS
Shibani	002	CSE	C, C++
Rupashree	003	EEE	C, DBMS

The above table is not in 1st Normal form as the attribute subject is not atomic. So the correct table in 1st Normal form will be:

[TABLE: Student]

Name	Roll	Branch	Subject
Jayashree	001	CSE	Java
Jayashree	001	CSE	C
Jayashree	001	CSE	DBMS
Shibani	002	CSE	C
Shibani	002	CSE	C++
Rupashree	003	EEE	C
Rupashree	003	EEE	DBMS

2nd Normal Form (2 NF): [Remove partial dependencies]

A relation schema R is in **second normal form** (2NF) if it is in 1NF and every non-primary key attribute is fully functionally dependent on the primary key.

Full and Partial Functional Dependency:

If A and B are attributes of a relation, A → B is **fully dependent** if some attribute is removed from A, then the dependency do not hold anymore. (or when all non-key attributes are dependent on the key attribute)

A → B is **partially dependent** if some attribute is removed from A and the dependency still holds.

3rd Normal Form (3 NF): [Remove transitive dependencies]

A relation that is in 1NF and 2NF, and in which no non-primary key attribute is transitively dependent on the primary key i.e. there do not exist any transitive dependencies.

or

A relation schema R is in **third normal form (3NF)** with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R.
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

Transitive dependency: A condition where A, B and C are attributes of a relation such that if A → B and B → C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).

Question 6: Given R(A, B, C, D, E, F, G,H) with FD (A →C, B →D,C →E, D →F, AB →G, G →E). Decompose R using 2NF and 3NF.

Solution:

Primary key is AB because $AB^+=R$

Using 2NF R is decomposed into

R1(A, C, E) with FDs F1(A→C, C→E)

R2(B, D, F) with FDs F2(B→D, D→F)

R3(AB, G, E) with FDs F3(AB→G, G→E)

Using 3NF R1 is decomposed into

R11(A, C) with FDs F1(A→C)

R12(A, E) with FDs F1(A→E)

Using 3NF R2 is decomposed into

R21(B, D) with FDs F1(B→D)

R22(B, F) with FDs F1(B→F)

Using 3NF R3 is decompose into

R31(A, B, G) with FDs F1(AB→G)

R32(A, B, E) with FDs F1(AB→ E)

Boyce–Codd Normal Form (BCNF)

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema R.

Decomposition using BCNF:

Find at least one non trivial functional dependency $\alpha \rightarrow \beta$, where α is not a super key. Then the relation R can be decomposed into

1. $\alpha \cup \beta$
2. $R - (\beta - \alpha)$

4th Normal Form (4 NF)

A relation schema R is in **fourth normal form** (4NF) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for schema R.

Multi-valued dependency:

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency** $\alpha \twoheadrightarrow \beta$ holds on R if, in any legal relation r(R), for all pairs of tuples t1 and t2 in r such that $t1[\alpha] = t2[\alpha]$, there exist tuples t3 and t4 in r such that:

$t1[\alpha] = t2[\alpha] = t3[\alpha] = t4[\alpha]$

$t3[\beta] = t1[\beta]$

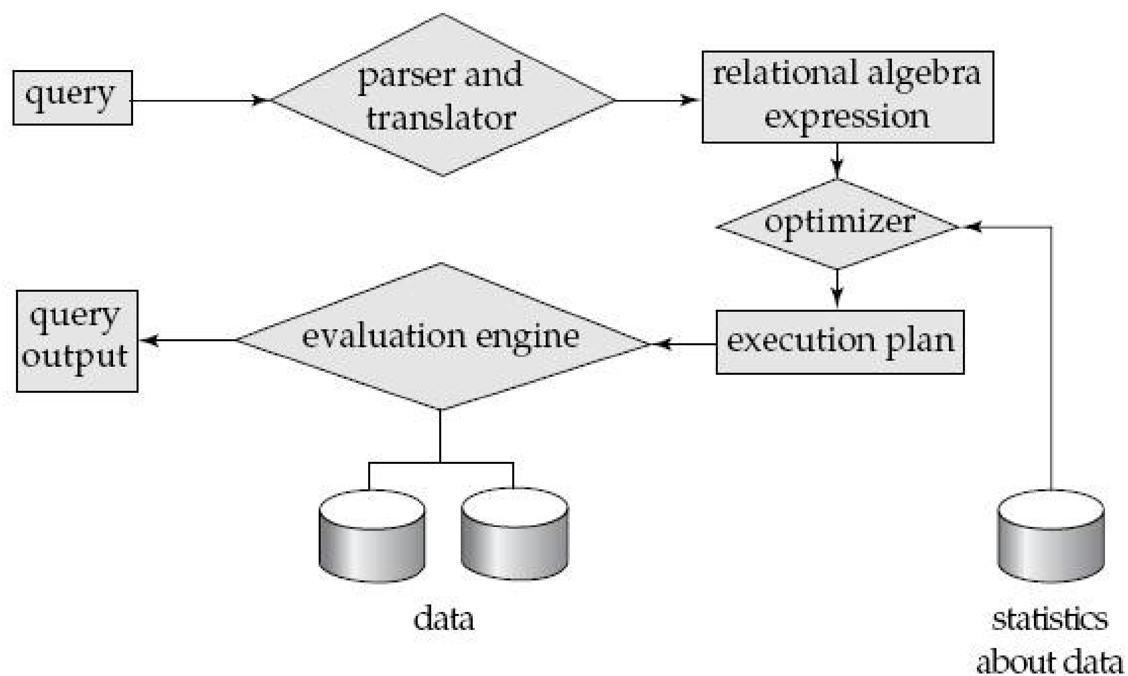
$t3[R - \beta] = t2[R - \beta]$

$t4[\beta] = t2[\beta]$

$t4[R - \beta] = t1[R - \beta]$

Query processing

- **Query processing** refers to the range of activities involved in extracting data from a database.
- The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.
- The steps involved in query processing are:
    1. Parsing and translation
    2. Optimization
    3. Evaluation



Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill-suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

As an illustration, consider the query

select balance

from account

where balance < 2500

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{balance<2500}$ (Πbalance (account))
- $\Pi_{balance}$ ($\sigma_{balance<2500}$ (account))

Transaction

- ✓ A **transaction** is a **unit** of program execution that accesses and updates various data items.
- ✓ Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java), where it is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**.
- ✓ The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

ACID Properties

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

1. Atomicity. A transaction is said to be atomic if all the operation of the transaction are executed at once or none of them is executed.

2. Consistency. A database is said to be consistent if a transaction is executed in a isolated manner.

3. Isolation. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

4. Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

Let Ti be a transaction that transfers 50 from account A to account B. This transaction can be defined as:

```
Read (A);
A := A – 50;
Write (A);
read(B);
B := B + 50;
Write (B).
```

read (X), which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
write (X), which transfers the data item X from the the local buffer of the transaction that executed the write back to the database.

Let us now consider each of the ACID requirements. (For ease of presentation, we consider them in an order different from the order A-C-I-D).

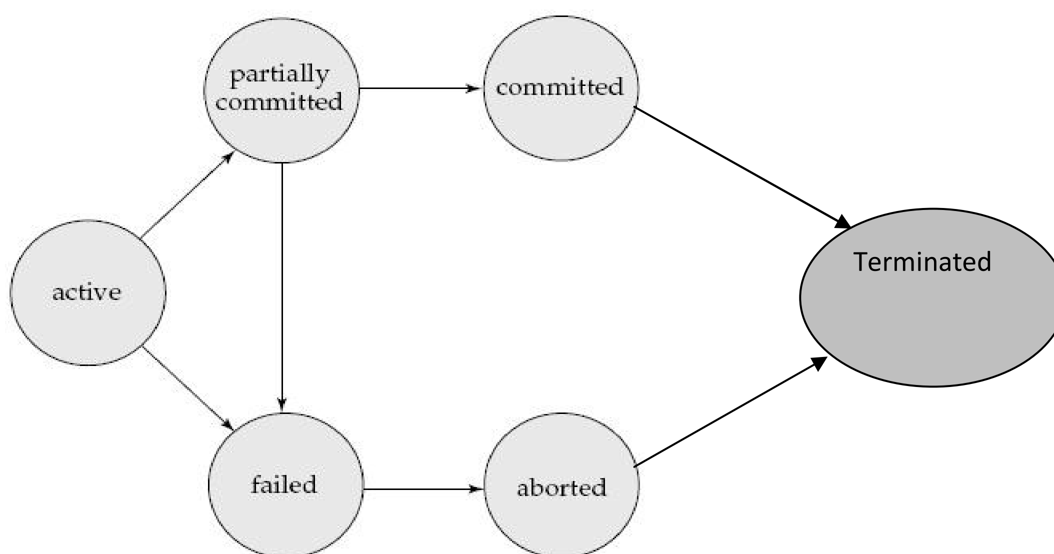
Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Atomicity: Suppose that, just before the execution of transaction Ti the values of accounts A and B are 1000 and 2000, respectively. Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully. Examples of such failures include **power failures, hardware failures,** and **software errors**.

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state. For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

State Diagram of a Transaction



Question 1: During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.

Answer: The possible sequences of states are:-

a. active→ partially committed →committed. This is the normal sequence a successful transaction will follow. After executing all its statements it enters the partially committed state. After enough recovery information has been written to disk, the transaction finally enters the committed state.

b. active → partially committed → aborted. After executing the last statement of the transaction, it enters the partially committed state. But before enough recovery information is written to disk, a hardware failure may occur destroying the memory contents. In this case the changes which it made to the database are undone, and the transaction enters the aborted state.

c. active →failed →aborted. After the transaction starts, if it is discovered at some point that normal execution cannot continue (either due to internal program errors or external errors), it enters the failed state. It is then rolled back, after which it enters the aborted state.

Concurrent Executions

- Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data.
- Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed.
- The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

- There are two good reasons for allowing concurrency:
    1. **Improved throughput and resource utilization**
    2. **Reduced waiting time**.

Conflict Serializability

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij, of transactions Ti and Tj , respectively (i _= j). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.

2. Ii = read(Q), Ij = write(Q). If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus, the order of Ii and Ij matters.

3. Ii = write(Q), Ij = read(Q). The order of Ii and Ij matters for reasons similar to those of the previous case.

4. Ii = write(Q), Ij = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write (Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

View Serializability

Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be **view equivalent** if three conditions are met:

1. For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S’, also read the initial value of Q.

2. For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj , then the read(Q) operation of transaction Ti must, in schedule S’, also read the value of Q that was produced by the same write(Q) operation of transaction Tj .

3. For each data item Q, the transaction (if any) that performs the final write (Q) operation in schedule S must perform the final write (Q) operation in schedule S’.

Question: Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

Answer: Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practise are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

Transaction Definition in SQL

A data-manipulation language must include a construct for specifying the set of actions that constitute a transaction. The SQL standard specifies that a transaction begins implicitly. Transactions are ended by one of these SQL statements:

- **Commit work** commits the current transaction and begins a new one.
- **Rollback work** causes the current transaction to abort.

There a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule S. Then construct a directed graph, called a **precedence graph**, from S. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of

vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. Ti executes write(Q) before T_j executes read(Q).

2. Ti executes read(Q) before T_j executes write(Q).

3. Ti executes write(Q) before T_j executes write(Q).

Concurrency Control

- Concurrent control is a technique to ensure that concurrent execution of transaction must results in a consistent database.
- It uses certain protocols to ensure the serializability of a schedule of concurrent executing transactions.
- Some of these protocols are:
  - ✓ Lock based protocol
  - ✓ Time stamp based protocol

Lock-Based Protocols

One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Lock

- It is a variable associated with a data item X that reflects the status of data item X with respect to possible operations that can be applied. A data item X can be locked in two modes:

  Shared lock

  If a transaction T has obtained shared mode lock, then T can read but cannot write. It is denoted by S.

  Exclusive lock

  If a transaction T has obtained exclusive mode lock, then T can both read and write. It is denoted by X.

- A transaction requests a shared lock on data item Q by executing the **lock-S (Q)** instruction. Similarly, a transaction requests an exclusive lock through the **lock-X (Q)** instruction. A transaction can unlock a data item Q by the **unlock (Q)** instruction.
- **Compatible Function:** Let A and B represent arbitrary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is **compatible** with mode B. Such a function can be represented conveniently by a matrix.
- An element comp (A, B) of the matrix has the value true if and only if mode A is compatible with mode B.

	S	**X**
S	True	False
X	False	False

- To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to **wait** until all incompatible locks held by other transactions have been released.
- **Example:**

T1:
lock-X(B);
read(B);
B := B – 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock (A).
	T2:
	lock-S(A);
	read(A);
	unlock(A);
	lock-S(B);
	read(B);
	unlock(B);
	display (A + B).

- Transaction Ti may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, for a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

T1	T2	Concurrency-Control Manager
lock-X(B)		
		grant-X(B, T1)
read(B)		
B := B 50		
write(B)		
unlock(B)	lock-S(A)	
		grant-S(A, T2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T2)
	read(B)	
	unlock(B)	
	display(A + B)	
lock-X(A)		grant-X(A, T2)
read(A)		
A := A + 50		
write(A)		
unlock(A)		

Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock.

Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock. At this point T2 may release the lock, but still T1 has to wait for T3 to finish. But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be starved.

The Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase. A transaction may obtain locks, but may not release any lock.

2. Shrinking phase. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

Timestamps base Protocol

Timestamps:

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj). There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if $TS(T_i) < TS(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j .

To implement this scheme, we associate with each data item Q two timestamp values:

• **W-timestamp** (Q) denotes the largest timestamp of any transaction that executed write (Q) successfully.

• **R-timestamp** (Q) denotes the largest timestamp of any transaction that executed read (Q) successfully.

These timestamps are updated whenever a new read (Q) or write (Q) instruction is executed.

DeadLock

- A system is in a deadlock state if there exist a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.

Deadlock Prevention:

Two different deadlock prevention schemes are:

wait–die

The **wait–die** scheme is a non-preemptive technique. When transaction Ti requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is, Ti is older than T_j). Otherwise, Ti is rolled back (dies).

For example, suppose that transactions T22, T23, and T24 have timestamps 5, 10, and 15, respectively. If T22 requests a data item held by T23, then T22 will wait. If T24 requests a data item held by T23, then T24 will be rolled back.

wound–wait

The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction Ti requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is, Ti is younger than T_j). Otherwise, T_j is rolled back (T_j is wounded by T_i).

Returning to our example, with transactions T22, T23, and T24, if T22 requests a data item held by T23, then the data item will be preempted from T23, and T23 will be rolled back. If T24 requests a data item held by T23, then T24 will wait.

Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Database Recovery System

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner.

Transaction failure. There are two types of errors that may cause a transaction to fail:

- **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
- **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.

System crash. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

Disk failure. A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recovers from the failure.

Storage Types

- **Volatile storage.** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
- **Nonvolatile storage.** Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage.
- **Stable storage.** Information residing in stable storage is never lost.

Storage Types

Database recovery is a technique that restores the database to the most recent consistent state that exists before the database failure.

Log-Based Recovery

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database.There are several types of log records. An **update log record** describes a single databasewrite. It has these fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as:

- <T_i start>. Transaction Ti has started.
- <T_i, X_j, V1, V2>. Transaction Ti has performed a write on data item X_j . X_j had value V1 before the write, and will have value V2 after the write.
- <T_i commit>. Transaction Ti has committed.
- <T_i abort>. Transaction Ti has aborted.

Deferred Database Modification

- The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits.

- When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.
- The execution of transaction Ti proceeds as follows. Before T_i starts its execution, a record <T_i start> is written to the log. A write(X) operation by T_i results in the writing of a new record to the log. Finally, when T_i partially commits, a record <T_i commit> is written to the log.

Immediate Database Modification

- The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called uncommitted modifications.
- Before a transaction Ti starts its execution, the system writes the record <T_i start> to the log. During its execution, any write(X) operation by T_i is preceded by the writing of the appropriate new update record to the log. When T_i partially commits, the system writes the record <T_i commit> to the log.