Database Management System

DBMS Notes

WWW.LEARNLONER.COM

Unit 1 - INTRODUCTION

Introduction to Database Management System

Data

- Fact that can be recorded or stored.
- E.g. Person Name, Age, Gender and Weight etc.

Information

- When data is processed, organized, structured or presented in a given context so as to make it useful, it is called information.

Database

- A Database is a collection of inter-related data.
- E.g. Books Database in Library, Student Database in University etc.

DBMS (Database Management System)

- A database management system is a collection of inter-related data and set of programs to manipulate those data.
- DBMS = Database + Set of programs
- E.g. MS SQL Server, Oracle, My SQL, SQLite, MongoDB etc.

Metadata

- Metadata is data about data.
- Data such as table name, column name, data type, authorized user and user access privileges for any table is called metadata for that table.

Data dictionary

- Data dictionary is an information repository which contains metadata.
- It is usually a part of the system catalog.

Data warehouse

- Data warehouse is an information repository which stored data.
- It is design to facilitate reporting and analysis.

Field

- A field is a character or group of characters that have a specific meaning.
- It is also called a data item. It is represented in the database by a value.
- For Example customer id, name, society and city are all fields for customer Data.

Record

- A record is a collection of logically related fields.

- For examples, collection of fields (id, name, address & city) forms a record for customer.

Database Management System (DBMS) and Its Applications:

A Database management system is a computerized record-keeping system. It is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow the users to define, store, retrieve and update the information contained in the database on demand. Information can be anything that is of significance to an individual or organization.

Databases touch all aspects of our lives. Some of the major areas of application are as follows:

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. Human resources

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- Add new students, instructors, and courses
- Register students for courses and generate class rosters
- Assign grades to students, compute grade point averages (GPA), and generate transcripts

System programmers wrote these application programs to meet the needs of the university.

New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major (say, computer science). As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, etc. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Keeping organizational information in a file processing system has a number of major disadvantages:

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 structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files).

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 student address may be reflected in the Music department records but not elsewhere in the system.

Difficulty in Accessing Data: 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.

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.

Integrity Problems. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. 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.

Atomicity Problems. A computer system, like any other 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 Rs. 5000 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the Rs. 5000 was removed from the balance of department *A* but was not credited to the balance of department *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.

Concurrent-Access Anomalies. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously.

Security Problems. Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

Advantages of DBMS:

Controlling of Redundancy: In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

Improved Data Sharing : DBMS allows a user to share the data in any number of application programs.

Data Integrity : Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can enforce an integrity that it must accept the customer only from Noida and Meerut city.

Security : Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.

Data Consistency: By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

Efficient Data Access: In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing.

Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.

2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.

3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.

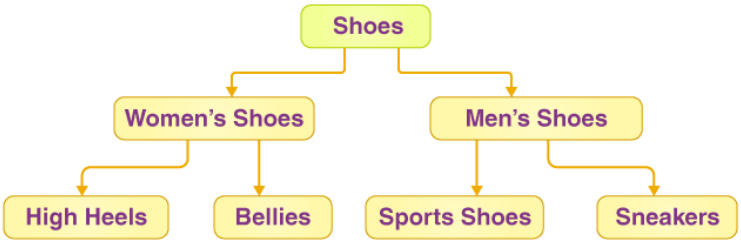
4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

Data Models

The Data Model gives us an idea of how the final system would look after it has been fully implemented. It specifies the data items as well as the relationships between them. In a database management system, data models are often used to show how data is connected, stored, accessed, and changed.

Hierarchical Model

The Hierarchical Model was the first database management system model. This concept uses a hierarchical tree structure to organize the data. The hierarchy begins at the root, which contains root data, and then grows into a tree as child nodes are added to the parent node. This model accurately depicts several real-world relationships such as food recipes, website sitemaps, and so on. The following diagram depicts the relationship between the shoes available on a shopping website:



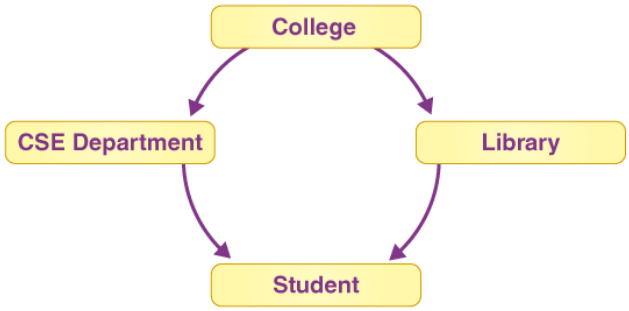
Pros n Cons of Hierarchical Model

- **Pros of Hierarchical Model**
  - A tree-like structure is incredibly straightforward and quick to navigate.
  - Any modification to the parent node is reflected automatically in the child node, ensuring data integrity.
- **Cons of Hierarchical Model**
  - Relationships that are complex are not supported.
  - Because it only supports one parent per child node, if we have a complex relationship in which a child node needs to have two parents, we won't be able to describe it using this model.
  - When a parent node is removed, the child node is removed as well.

Network Model

Like the hierarchical data model, this model is fairly simple and easy to create. It's intended to be a flexible way of representing items and their interactions. The main difference between this model and the hierarchical model is that any record can have several parents. It uses a graph instead of a hierarchical tree. The hierarchical model is extended in the

network model. It has entities that are grouped in a graphical format, and some of the entities can be reached by many paths.



Network Model

Features of Network Model:

- **Multiple Paths:** There may be several paths to the same record due to the increased number of relationships. It allows for quick and easy data access.
- **The Ability to Merge More Relationships:** Data is more connected in this model since there are more relationships. This paradigm can handle many-to-many as well as one-to-one relationships.
- **Circular Linked List:** The circular linked list is used to perform operations on the network model. The present position is kept up to date with the help of a software, and it navigates through the records based on the relationship.

Pros n Cons of Hierarchical Model

- **Pros of Network Model**
  - In comparison to the hierarchical model, data can be retrieved faster. This is because the data in the network model is more related, and there may be more than one path to a given node. As a result, the data can be accessed in a variety of ways.
  - Data integrity is present since there is a parent-child relationship. Any changes to the parent record are mirrored in the child record.
- **Cons of Network Model**
  - As the number of relationships to be managed grows, the system may get increasingly complicated. To operate with the model, a user must have a thorough understanding of it.
  - Any alteration, such as an update, deletion, or insertion, is extremely difficult.

Relational Model Concepts

The relational model represents the database as a collection of *relations.* Informally, each relation resembles a **table** of values or file of records. The table name and column names are used to help to interpret the meaning of the values in each row. All values in a column are of the same data type.
In the formal relational model terminology, a row is called a *tuple,* a column header is called an *attribute,* and the table is called a *relation.* The data type describing the types of values that can appear in each column is represented by a *domain* of possible values. We now define these terms—*domain, tuple, attribute,* and *relation*— formally.

Domains, Attributes, Tuples, and Relations

A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. Some examples of domains follow:

- ✓ *Mobile_Numbers*: The set of ten-digit phone numbers valid in India.
- ✓ *Adhar_No:* The set of valid twelve-digit Unique Identification Numbers.
- ✓ *Names*: The set of character strings that represent names of persons.
- ✓ *Employe_ Ages*: Possible ages of employees in a company; each must be an integer value between 18 and 65.

The preceding are called *logical* definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain Mobile_Numbers can be declared as a character string of the form *ddddddddddd*, where each *d* is a numeric (decimal) digit. The data type for Employee_ages is an integer number between 18 and 65. A domain is thus given a name, data type, and format.

Popular Relational Database Management Systems:

- IBM – DB2 and Informix Dynamic Server
- Oracle – Oracle and RDB
- Microsoft – SQL Server and Access

Properties of a Relational Model

The relational databases consist of the following properties:

- Every row is unique
- All of the values present in a column hold the same data type
- Values are atomic
- The columns sequence is not significant
- The rows sequence is not significant

- The name of every column is unique

Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

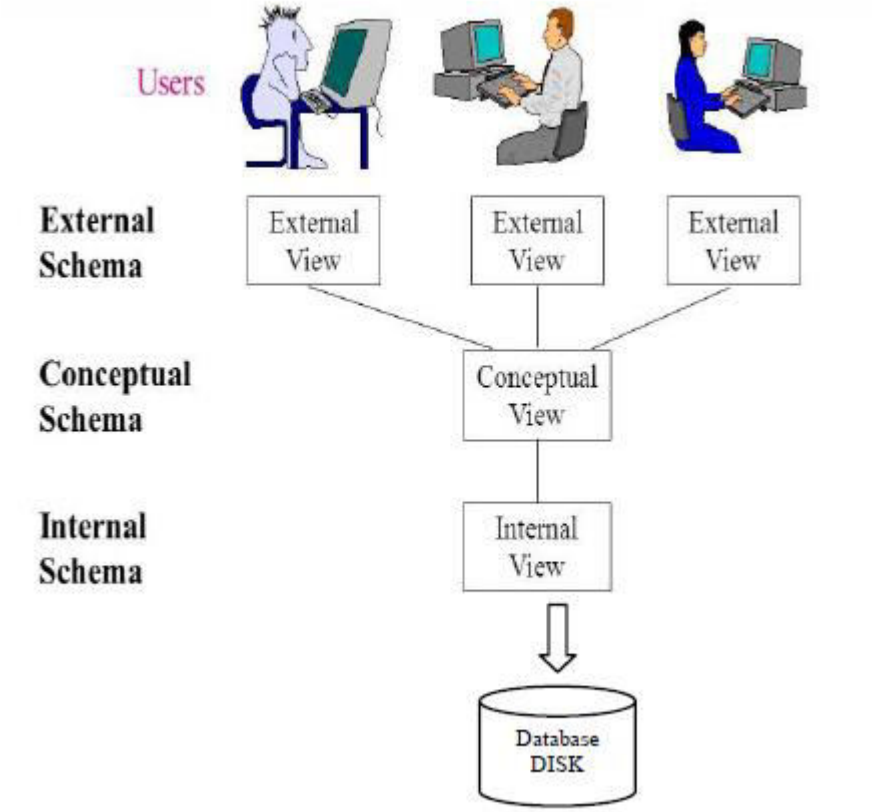


Figure : Levels of Abstraction in a DBMS

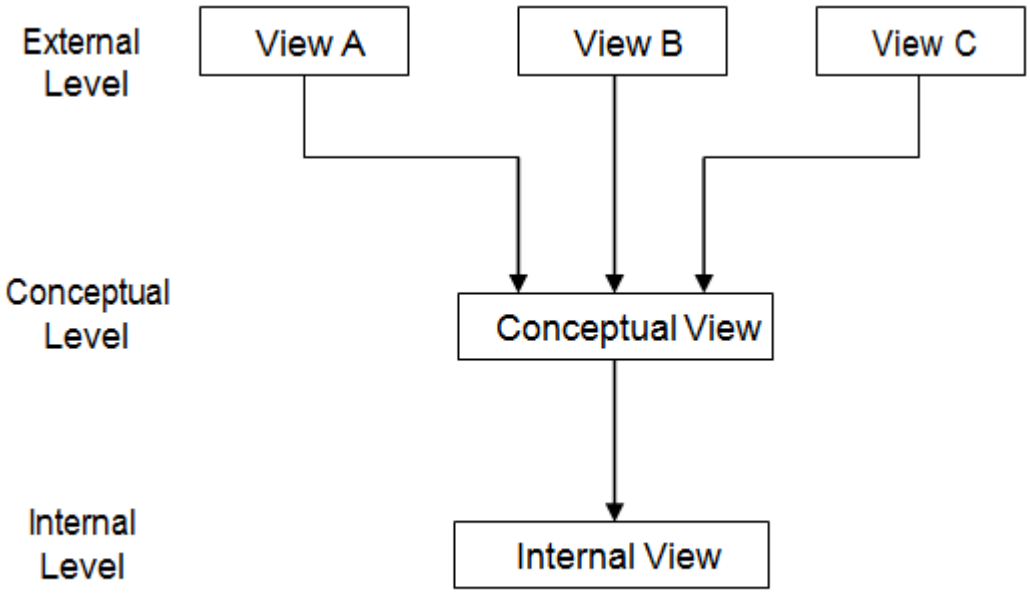
• **Physical level (or Internal View / Schema)**: The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

• **Logical level (or Conceptual View / Schema)**: 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 (or External View / Schema):** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database. Figure shows the relationship among the three levels of abstraction.

The ANSI SPARC architecture divided into three levels:

1. *External level*
2. *Conceptual level*
3. *Internal level*



Three levels ANSI SPARC Database System

Internal Level

- ✓ This is the lowest level of the data abstraction.
- ✓ It describes **how** the data are actually stored on storage devices.
- ✓ It is also known as a **physical level**.
- ✓ The internal view is described by internal schema.
- ✓ Internal schema consists of definition of stored record, method of representing the data field and access method used.

Conceptual Level

- ✓ This is the next higher level of the data abstraction.
- ✓ It describes what data are stored in the database and what relationships exist among those data.
- ✓ It is also known as a logical level.
- ✓ Conceptual view is defined by conceptual schema. It describes all records and relationship.

External Level

- ✓ This is the highest level of data abstraction.
- ✓ It is also known as view level.
- ✓ It describes only part of the entire database that a particular end user requires.
- ✓ External view is describes by external schema.
- ✓ External schema consists of definition of logical records, relationship in the external view and method of deriving the objects from the conceptual view.
- ✓ This object includes entities, attributes and relationship.

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, which describe different views of the database.

Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller. The typical user interface for naive users is a forms interface, where the user can fill in

ppropriate fields of the form. Naive users may also simply read *reports* generated from the database.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.

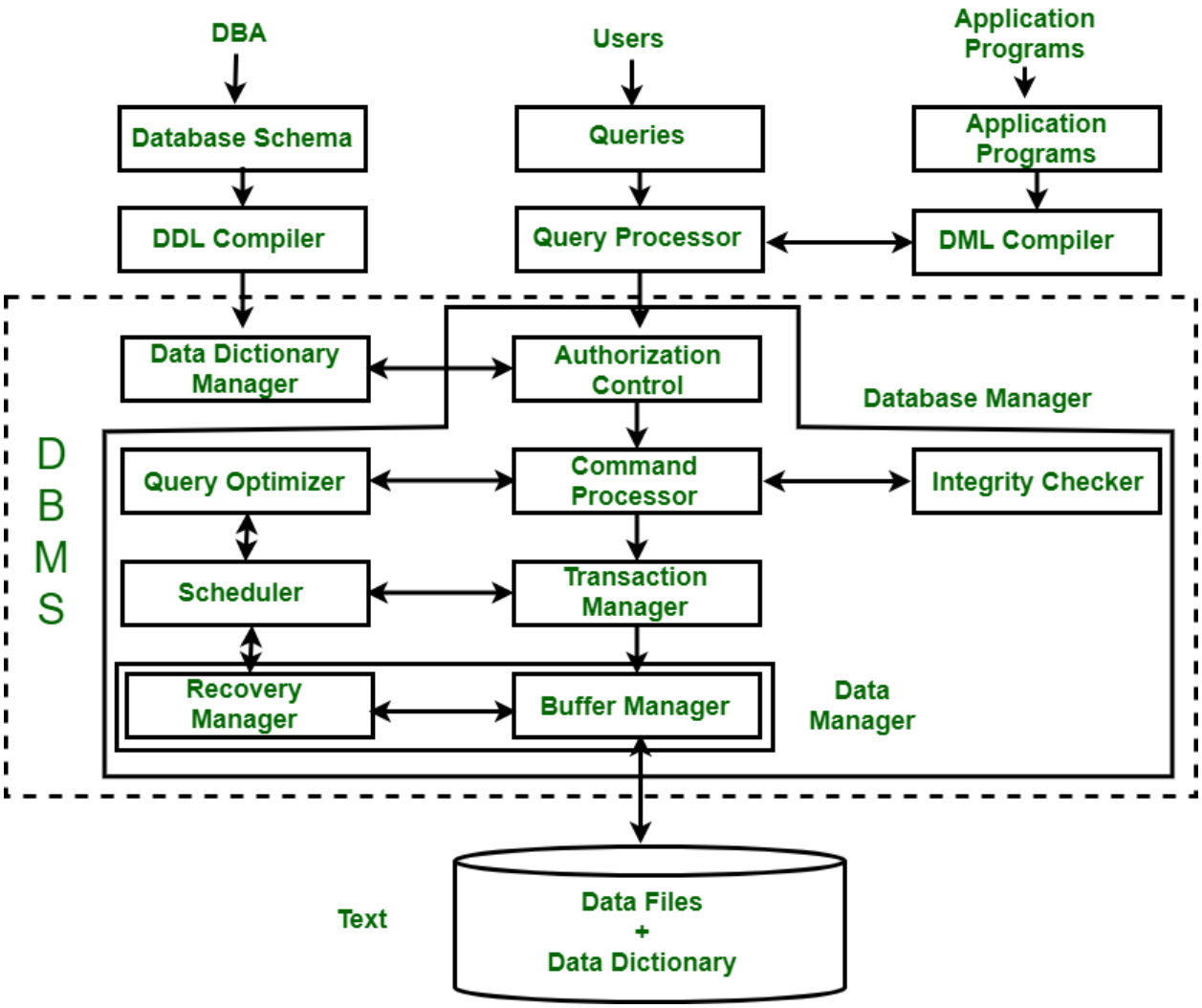
Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Database Administrator

- A Database Administrator (DBA) is an individual or person responsible for controlling, maintaining, coordinating, and operating a database management system. Managing, securing, and taking care of the database systems is a prime responsibility.
- They are responsible and in charge of authorizing access to the database, coordinating, capacity, planning, installation, and monitoring uses, and acquiring and gathering software and hardware resources as and when needed.
- Their role also varies from configuration, database design, migration, security, troubleshooting, backup, and data recovery. Database administration is a major and key function in any firm or organization that is relying on one or more databases. They are overall commanders of the Database system.
- The database administrator is a person in the organization who controls the design and the use of the database.
- DBA provides necessary technical support for implementing a database. DBA is involved more in the design, development, testing and operational phases.

Database System Architecture



Components of a DBMS

These functional units of a database system can be divided into two parts:

1. Query Processor Units(Components)
2. Storage Manager Units

Query Processor Units:

Query processor unit deal with execution of DDL (Data Definition Language) and DML (Data Manipulation Language) statements.

- **DDL Interpreter —** Interprets DDL statements into a set of tables containing metadata.
- **DML Compiler —** Translates DML statements into low level instructions that the query evaluation engine understands.
- **Embedded DML Pre-compiler —** Converts DML statements embedded in an application program into normal procedure calls in the host language.
- **Query Evaluation Engine —** Executes low level instructions generated by DML compiler.

Storage Manager Units:

Storage manager units provide interface between the low level data stored in database and the application programs & queries submitted to the system.

- **Authorization Manager** — Checks the authority of users to access data.
- **Integrity Manager** — Checks for the satisfaction of the integrity constraints.
- **Transaction Manager** — Preserves atomicity and controls concurrency.
- **File Manager** — Manages allocation of space on disk storage.
- **Buffer Manager** — Fetches data from disk storage to memory for being used.

In addition to these functional units, several data structures are required to implement physical storage system. These are described below:

- **Data Files** — To store user data.
- **Data Dictionary and System Catalog** — To store metadata. It is used heavily, almost for each and every data manipulation operation. So, it should be accessed efficiently.
- **Indices** — To provide faster access to data items.
- **Statistical Data** — To store statistical information about the data in the database. This information is used by the query processor to select efficient ways to execute a query.

Data Independence

Types of Data Independence

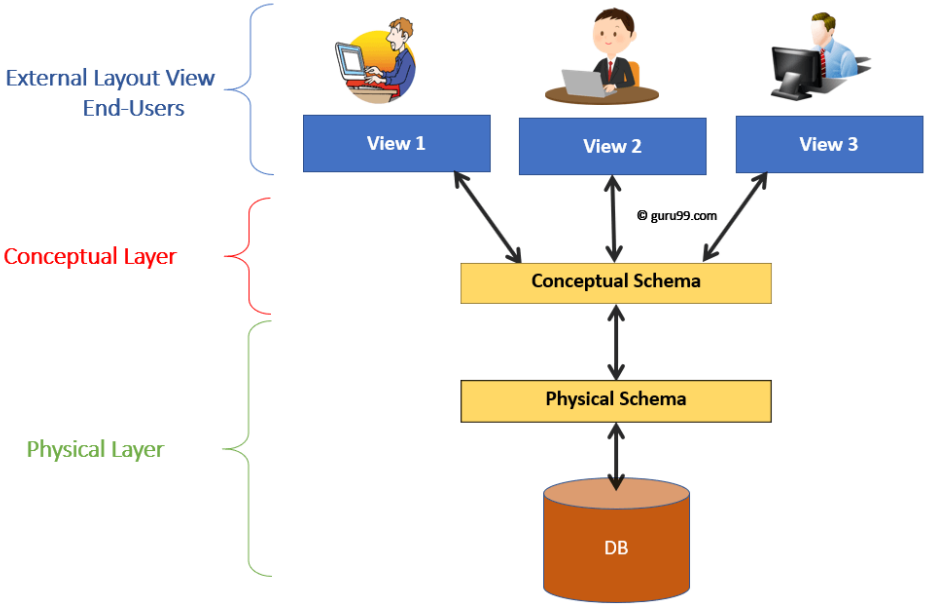
In DBMS there are two types of data independence

1. Physical data independence
2. Logical data independence.

Levels of Database

Before we learn Data Independence, a refresher on Database Levels is important. The database has 3 levels as shown in the diagram below

1. Physical/Internal
2. Conceptual
3. External



Levels of DBMS Architecture Diagram

Consider an Example of a University Database. At the different levels this is how the implementation will look like:

Type of Schema	Implementation
External Schema	**View 1**: Course info(cid:int,cname:string) **View 2**: studeninfo(id:int. name:string)

Type of Schema	Implementation

Conceptual Shema	`Students(id: int, name: string, login: string, age: integer)`
	`Courses(id: int, cname:string, credits:integer)`
	`Enrolled(id: int, grade:string)`

Physical Schema	• Relations stored as unordered files. • Index on the first column of Students.

Physical Data Independence

Physical data independence helps you to separate conceptual levels from the internal / physical levels. It allows you to provide a logical description of the database without the need to specify physical structures. Compared to Logical Independence, it is easy to achieve physical data independence.

With Physical independence, you can easily change the physical storage structures or devices without an effect on the conceptual schema. Any change done would be absorbed by the mapping between the conceptual and internal levels.

Examples of changes under Physical Data Independence

Due to Physical independence, any of the below change will not affect the conceptual layer.

- Using a new storage device like Hard Drive or Magnetic Tapes
- Modifying the file organization technique in the Database
- Switching to different data structures.
- Changing the access method.
- Modifying indexes.
- Changes to compression techniques or hashing algorithms.
- Change of Location of Database from say C drive to D Drive

Logical Data Independence

Logical Data Independence is the ability to change the conceptual scheme without changing

1. External views
2. External API or programs

Any change made will be absorbed by the mapping between external and conceptual levels.

When compared to Physical Data independence, it is challenging to achieve logical data independence.

Examples of changes under Logical Data Independence

Due to Logical independence, any of the below change will not affect the external layer.

1. Add/Modify/Delete a new attribute, entity or relationship is possible without a rewrite of existing application programs
2. Merging two records into one
3. Breaking an existing record into two or more records

Difference between Physical and Logical Data Independence

Logical Data Independence	Physical Data Independence
Logical Data Independence is mainly concerned with the structure or changing the data definition.	Mainly concerned with the storage of the data.
It is difficult as the retrieving of data is mainly dependent on the logical structure of data.	It is easy to retrieve.
Compared to Logic Physical independence it is difficult to achieve logical data independence.	Compared to Logical Independence it is easy to achieve physical data independence.
You need to make changes in the Application program if new fields are added or deleted from the database.	A change in the physical level usually does not need change at the Application program level.
Modification at the logical levels is significant whenever the logical structures of the database are changed.	Modifications made at the internal levels may or may not be needed to improve the performance of the structure.
Concerned with conceptual schema	Concerned with internal schema
Example: Add/Modify/Delete a new attribute	Example: change in compression techniques, hashing algorithms, storage devices, etc.

Importance of Data Independence

- Helps you to improve the quality of the data
- Database system maintenance becomes affordable
- Enforcement of standards and improvement in database security

- You don’t need to alter data structure in application programs
- Permit developers to focus on the general structure of the Database rather than worrying about the internal implementation
- It allows you to improve state which is undamaged or undivided
- Database incongruity is vastly reduced.
- Easily make modifications in the physical level is needed to improve the performance of the system.

Entity – Relationship Model

An **entity–relationship model (ER model)** is a systematic way of describing and defining a business process. An ER model is typically implemented as a database. The main components of E-R model are: entity set and relationship set.

E-R Diagram

ER diagrams are used to represent the E-R model in a database, which makes them easy to be converted into relations (tables). ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.

Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

- **Rectangles:** Rectangles represent Entities in ER Model.
- **Ellipses:** Ellipses represent Attributes in ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent Multi-Valued Attributes.
- **Double Rectangle:** Double Rectangle represents a Weak Entity.

Symbols used in ER Diagram

Entity

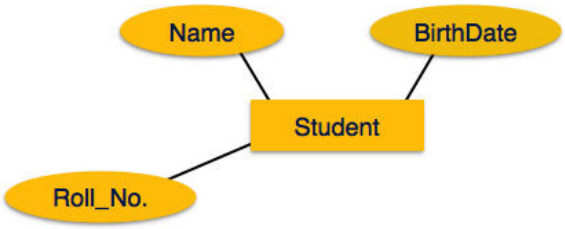
An entity can be a real-world object, that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity.

Entity Set

An entity set is a collection of similar types of entities. An entity set may contain entities with attribute sharing similar values. For example, a Students set may contain all the students of a school; likewise a Teachers set may contain all the teachers of a school from all faculties. Entity sets need not be disjoint.

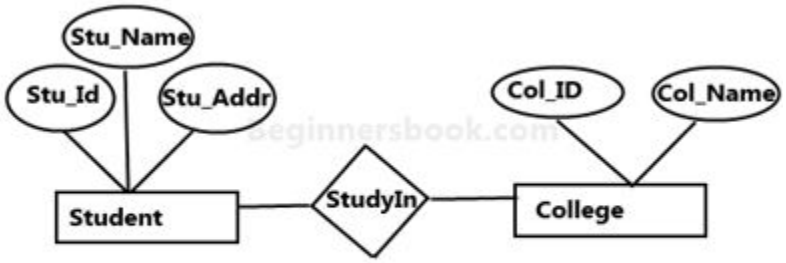
Attributes

Entities are represented by means of their properties, called attributes. All attributes have values. For example, a student entity may have name, class, and age as attributes.



There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

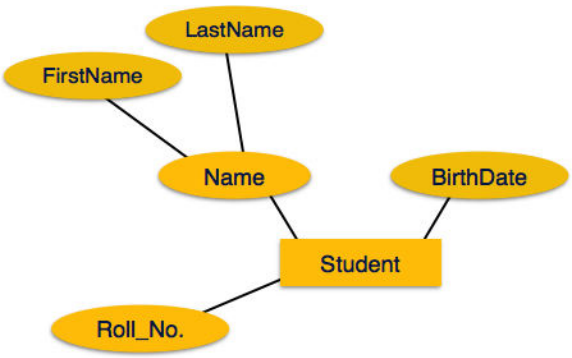
A sample E-R Diagram:



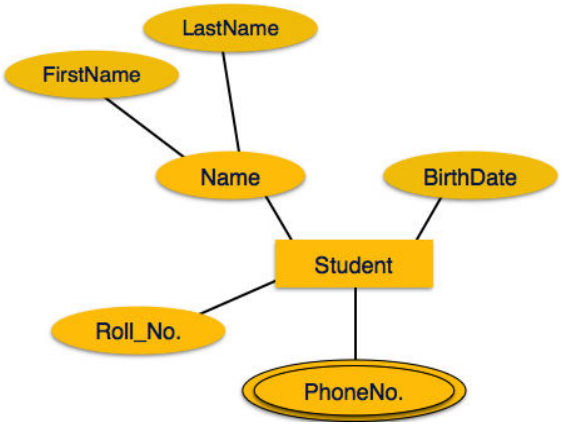
Sample E-R Diagram

Types of Attributes

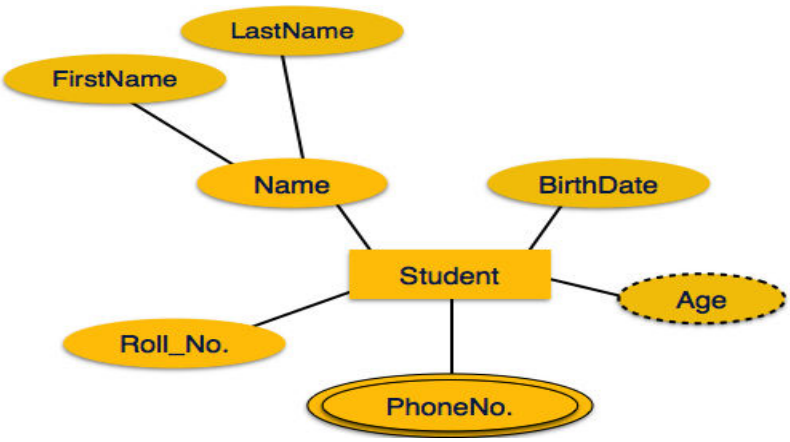
- **Simple attribute** − Simple attributes are atomic values, which cannot be divided further. For example, a student's phone number is an atomic value of 10 digits
- **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name



- **Single-value attribute** − Single-value attributes contain single value. For example − Social_Security_Number.
- **Multivalued Attributes**: An attribute that can hold multiple values is known as multivalued attribute. We represent it with double ellipses in an E-R Diagram. E.g. A person can have more than one phone numbers so the phone number attribute is multivalued.

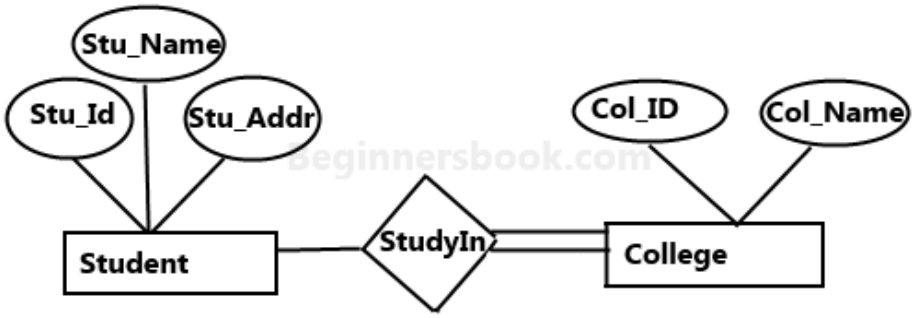


- **Derived Attribute:** A derived attribute is one whose value is dynamic and derived from another attribute. It is represented by dashed ellipses in an E-R Diagram. E.g. Person age is a derived attribute as it changes over time and can be derived from another attribute (Date of birth).



Total Participation of an Entity set:

A Total participation of an entity set represents that each entity in entity set must have at least one relationship in a relationship set. For example: In the below diagram each college must have at-least one associated Student.



Entity-Set and Keys

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

For example, the roll_number of a student makes him/her identifiable among students.

- **Super Key** − A set of attributes (one or more) that collectively identifies an entity in an entity set.
- **Candidate Key** − A minimal super key is called a candidate key. An entity set may have more than one candidate key.
- **Primary Key** − A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.

Relationship

The association among entities is called a relationship. For example, an employee works_at a department, a student enrolls in a course. Here, Works_at and Enrolls are called relationships.

Relationship Set

A set of relationships of similar type is called a relationship set. Like entities, a relationship too can have attributes. These attributes are called descriptive attributes.

Degree of Relationship

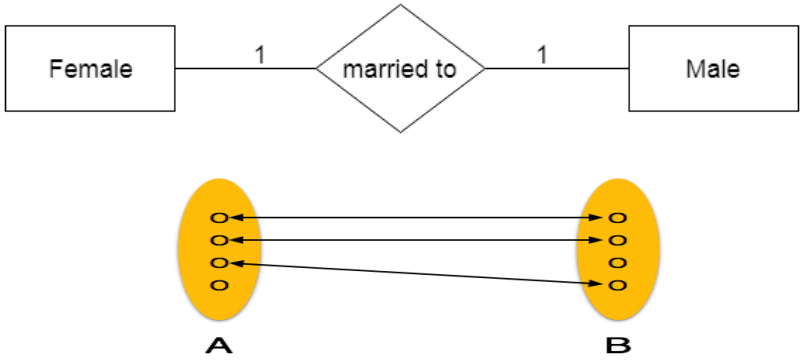
The number of participating entities in a relationship defines the degree of the relationship.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

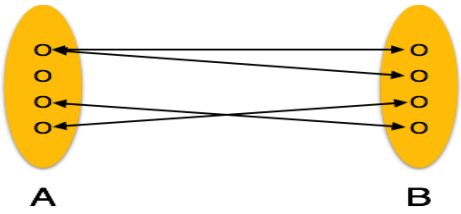
Mapping Cardinalities

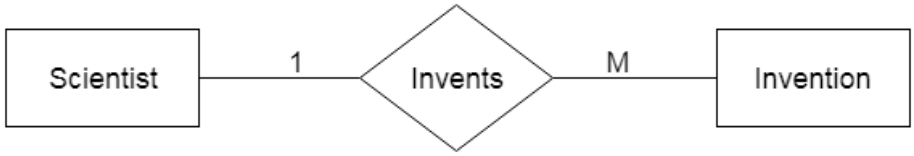
Cardinality defines the number of entities in one entity set, which can be associated with the number of entities of other set via relationship set.

- One-to-one − One entity from entity set A can be associated with at most one entity of entity set B and vice versa.

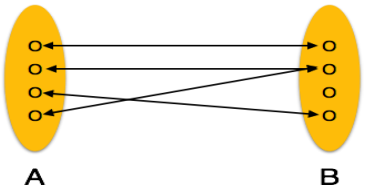


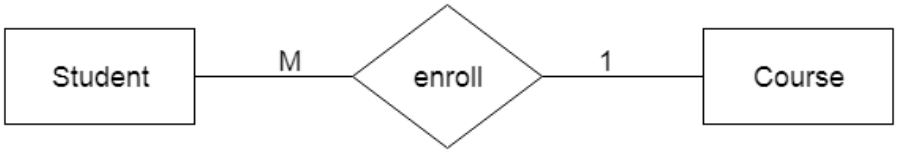
- **One-to-many** − One entity from entity set A can be associated with more than one entities of entity set B however an entity from entity set B, can be associated with at most one entity.



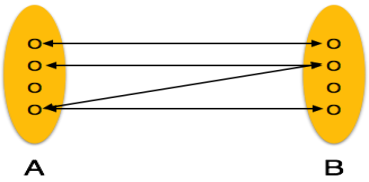


- **Many-to-one** − More than one entities from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A





- **Many-to-many** − One entity from A can be associated with more than one entity from B and vice versa.

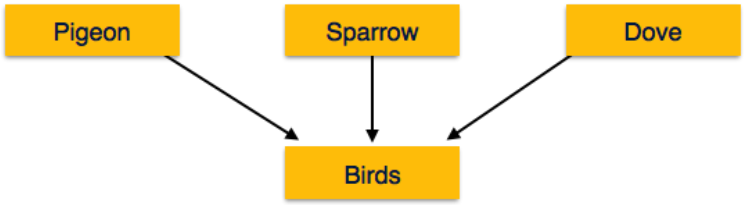


Extended E-R Features

Generalization

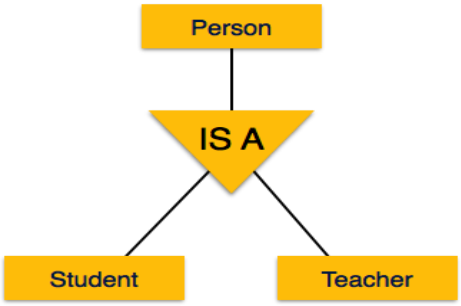
The ER Model has the power of expressing database entities in a conceptual hierarchical manner. As the hierarchy goes up, it generalizes the view of entities, and as we go deep in the hierarchy, it gives us the detail of every entity included.

Going up in this structure is called **generalization**, where entities are clubbed together to represent a more generalized view. The process of generalizing entities, where the generalized entities contain the properties of all the generalized entities is called generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



Specialization

Specialization is the opposite of generalization. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group 'Person' for example. A person has name, date of birth, gender, etc. These properties are common in all persons, human beings. But in a company, persons can be identified as employee, employer, customer, or vendor, based on what role they play in the company.



Similarly, in a school database, persons can be specialized as teacher, student, or a staff, based on what role they play in school as entities.

Constraints in DBMS

Constraints enforce limits to the data or type of data that can be inserted / updated / deleted from a table. The whole purpose of constraints is to maintain the **data integrity** during an update/delete/insert into a table.

Types of constraints

- NOT NULL
- UNIQUE
- DEFAULT
- CHECK
- Key Constraints – PRIMARY KEY, FOREIGN KEY
- Domain constraints
- Mapping constraints

NOT NULL:

NOT NULL constraint makes sure that a column does not hold NULL value. When we don't provide value for a particular column while inserting a record into a table, it takes NULL value by default. By specifying NULL constraint, we can be sure that a particular column(s) cannot have NULL values

UNIQUE:

UNIQUE Constraint enforces a column or set of columns to have unique values. If a column has a unique constraint, it means that particular column cannot have duplicate values in a table

DEFAULT:

The DEFAULT constraint provides a default value to a column when there is no value provided while inserting a record into a table.

CHECK:

This constraint is used for specifying range of values for a particular column of a table. When this constraint is being set on a column, it ensures that the specified column must have the value falling in the specified range.

Key constraints:

PRIMARY KEY:

Primary key uniquely identifies each record in a table. It must have unique values and cannot contain nulls. In the below example the ROLL_NO field is marked as primary key, that means the ROLL_NO field cannot have duplicate and null values.

FOREIGN KEY:

Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

DOMAIN CONSTRAINTS:

Each table has certain set of columns and each column allows a same type of data, based on its data type. The column does not accept values of any other data type.

Unit 2 – RELATIONAL MODEL

Structure of Relational Databases:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure-1, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept_name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept_name*, and *salary*. Similarly, the *course* table of Figure-2 stores information about courses, consisting of a *course id*, *title*, *dept_name*, and *credits*, for each course.

Figure-3 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course id* and *prereq id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other. As another example, we consider the table *instructor*, a row in the table can be thought of as representing the relationship between a specified *ID* and the corresponding values for *name*, *dept_name,* and *salary* values.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure-1: The *instructor* relation

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure-2: The *course* relation

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure-3: The *prereq* relation.

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure-1, we can see that the relation *instructor* has four attributes:

ID*, *name*, *dept name*, and *salary

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, or are unsorted, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we will mostly show the relations sorted by their first attribute.

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Figure: Unsorted display of the *instructor* relation.

For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units.

The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone_number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist.

Database Schema

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure: The *department* relation.

In general, a relation schema consists of a list of attributes and their corresponding domains.

Consider the *department* relation of Figure. The schema for that relation is

***department* (*dept_name*, *building*, *budget*)**

Each course in a university may be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individual offering, or section, of the class. The schema is

***section* (*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)**

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure: The *section* relation.

Figure shows a sample instance of the *section* relation. We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is

***teaches* (*ID*, *course id*, *sec id*, *semester*, *year*)**

Figure shows a sample instance of the *teaches* relation. As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *instructor*, *department*, *course*, *section*, *prereq*, and *teaches*, we use the following relations in this unit:

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure: The *teaches* relation.

- *student* (*ID*, *name*, *dept_name*, *tot_cred*)
- *advisor* (*s_id*, *i_id*)
- *takes* (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
- *classroom* (*building*, *room number*, *capacity*)
- *time slot* (*time_slot_id*, *day*, *start_time*, *end_time*)

Keys

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If *K* is a superkey, then so is any superset of *K*. We are often interested in super keys for which no proper subset is a superkey. Such minimal super keys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept_name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and *{name*, *dept_name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID*, *name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined. A relation, say r_1, may include among its attributes the primary key of another relation, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2.

The relation r_1 is also called the **referencing relation** of the foreign key dependency, and r_2 is called the **referenced relation** of the foreign key. For example, the attribute *dept_name* in *instructor* is a foreign key from *instructor*, referencing *department*, since *dept_name* is the primary key of *department*.

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one instructor; however, it could possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

Relational Algebra and Relational Calculus

A data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining the database's structure and constraints. The basic set of operations for the relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as *relational algebra expressions*. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries. A relational calculus expression creates a new relation. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus.

The relational algebra is often considered to be an integral part of the relational data model. Its operations can be divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the *formal* relational model. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT (also known as CROSS PRODUCT). The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others.

Some common database requests cannot be performed with the original relational algebra operations, so additional operations were created to express these requests. These include **aggregate functions**, which are operations that can *summarize* data from the tables, as well as additional types of JOIN and UNION operations, known as OUTER JOINs and OUTER UNIONs.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

The COMPANY relational database to be used for examples.

There are two variations of relational calculus. The *tuple* relational calculus and the *domain* relational calculus. In tuple relational calculus, variables range over *tuples*, whereas in domain relational calculus, variables range over the *domains* (values) of attributes.

Unary Relational Operations:

SELECT and PROJECT

The SELECT Operation

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**. One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$\sigma_{Dno=4}(EMPLOYEE)$

$\sigma_{Salary>30000}(EMPLOYEE)$

In general, the SELECT operation is denoted by

$\sigma_{<selection\ condition>}(R)$

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*. Notice that *R* is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as *R*.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<*attribute name*> <*comparison op*> <*constant value*>

or

<*attribute name*> <*comparison op*> <*attribute name*>

where <attribute name> is the name of an attribute of *R*, <comparison op> is normally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant value from the attribute domain. Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition. For example, to select the tuples for all employees

who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

$$\sigma_{(Dno=4\ \textbf{AND}\ Salary>25000)\ \textbf{OR}\ (Dno=5\ \textbf{AND}\ Salary>30000)}(EMPLOYEE)$$

The result is shown in Figure (a).

Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Hence, a sequence of SELECTs can be applied in any order.

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{Lname,\ Fname,\ Salary}(EMPLOYEE)$$

The resulting relation is shown in Figure (b). The general form of the PROJECT operation is

$$\pi_{<attribute\ list>}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sub list of attributes from the attributes of relation *R*. If the attribute list includes only nonkey attributes of *R*, duplicate tuples are likely to occur. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**. For example, consider the following PROJECT operation:

$$\pi_{Sex,\ Salary}(EMPLOYEE)$$

The result is shown in Figure (c). Notice that the tuple <'F', 25000> appears only once in Figure (c), even though this combination of values appears twice in the EMPLOYEE relation.

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Sequences of Operations and the RENAME Operation

In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results. For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

$$\pi_{Fname,\ Lname,\ Salary}(\sigma_{Dno=5}(EMPLOYEE))$$

Figure 2(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$

$$RESULT \leftarrow \pi_{Fname,\ Lname,\ Salary}(DEP5\_EMPS)$$

To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$TEMP \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$

$$R(First\_name,\ Last\_name,\ Salary) \leftarrow \pi_{Fname,\ Lname,\ Salary}(TEMP)$$

These two operations are illustrated in Figure 2(b).

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation *R* of degree *n* is denoted by any of the following three forms:

$$\rho_{S(B1, B2, ..., Bn)}(R) \text{ or } \rho_{S}(R) \text{ or } \rho_{(B1, B2, ..., Bn)}(R)$$

where the symbol ρ (rho) is used to denote the RENAME operator, *S* is the new relation name, and *B*1, *B*2, ..., *Bn* are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of *R* are (*A*1, *A*2, ..., *An*) in that order, then each *Ai* is renamed as *Bi*.

(a)

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figure 2
Results of a sequence of operations. (a) $\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$. (b) Using intermediate relations and renaming of attributes.

(b)

TEMP

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston,TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble,TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Relational Algebra Operations from Set Theory

The UNION, INTERSECTION, and MINUS Operations

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**). These are **binary** operations; that is, each is applied to two sets (of tuples).When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility* or *type compatibility*. Two relations $R(A1, A2, ..., An)$ and $S(B1, B2, ..., Bn)$ are said to be **union compatible** (or **type compatible**) if they have the same degree n and if $dom(Ai) = dom(Bi)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

■ **UNION:** The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

■ **INTERSECTION:** The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.

■ **SET DIFFERENCE** (or **MINUS**): The result of this operation, denoted by $R – S$, is a relation that includes all tuples that are in R but not in S.

Figure illustrates the three operations. Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R \text{ and } R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are also *associative operations;* that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \text{ and } (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is *not commutative;* that is, in general,

$$R - S \neq S - R$$

Figure (d) shows the names of students who are not instructors, and Figure (e) shows the names of instructors who are not students.

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

$$R \cap S = ((R \cup S) - (R - S)) - (S - R)$$

The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

CARTESIAN PRODUCT operation is also known as **CROSS PRODUCT** or **CROSS JOIN** denoted by ×. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A1, A2, ..., An) \times S(B1, B2, ..., Bm)$ is a relation Q with degree $n + m$ attributes $Q(A1, A2, ..., An, B1, B2, ..., Bm)$, in that order.

The resulting relation *Q* has one tuple for each combination of tuples—one from *R* and one from *S*. Hence, if *R* has *nR* tuples (denoted as |*R*| = *nR*), and *S* has *nS* tuples, then *R* × *S* will have *nR* * *nS* tuples.

The *n*-ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from *n* underlying relations.

In general, the CARTESIAN PRODUCT operation applied by itself is generally meaningless.

It is mostly useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex='F'}}(\text{EMPLOYEE})$$

$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$

$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

The resulting relations from this sequence of operations are shown in Figure.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Fig. Cartesian Product

Binary Relational Operations: JOIN and DIVISION

The JOIN Operation

The **JOIN** operation, denoted by ⋈, is used to combine *related tuples* from two relations into single "longer" tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations.

To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple. We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

$$DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$$

$$RESULT \leftarrow \pi_{Dname,\ Lname,\ Fname}(DEPT\_MGR)$$

The first operation is illustrated in Figure 6. Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used very frequently when specifying database queries. Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$$EMP\_DEPENDENTS \leftarrow EMPNAMES \times DEPENDENT$$

$$ACTUAL\_DEPENDENTS \leftarrow \sigma_{Ssn=Essn}(EMP\_DEPENDENTS)$$

These two operations can be replaced with a single JOIN operation as follows:

$$ACTUAL\_DEPENDENTS \leftarrow EMPNAMES \bowtie_{Ssn=Essn} DEPENDENT$$

The general form of a JOIN operation on two relations $R(A1, A2, ..., An)$ and $S(B1, B2, ..., Bm)$ is

$$R \bowtie_{<join\ condition>} S$$

The join condition is specified on attributes from the two relations *R* and *S* and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation *Q as a single combined tuple*.

A general join condition is of the form

<*condition*> ***AND*** <*condition*> ***AND***...***AND*** <*condition*>

where each <condition> is of the form $Ai\ \theta\ Bj$, *Ai* is an attribute of *R*, *Bj* is an attribute of *S*, *Ai* and *Bj* have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}. A JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of

the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

Figure 6
Result of the JOIN operation DEPT_MGR ← DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE.

The EQUIJOIN and NATURAL JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. For example, in Figure 6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN** denoted by ***** was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first. Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

$$PROJ\_DEPT \leftarrow PROJECT * \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$

The same query can be done in two steps by creating an intermediate table DEPT as follows:

$$DEPT \leftarrow \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$
$$PROJ\_DEPT \leftarrow PROJECT * DEPT$$

The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 7(a). In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the

DEPARTMENT tuple for the department that controls the project, but *only one join attribute value* is kept.

(a)

PROJ_DEPT

Pname	Pnumber	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 7
Results of two NATURAL JOIN operations. (a) PROJ_DEPT ← PROJECT * DEPT.
(b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

As we can see, a single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called *outer join*. Informally, an *inner join* is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. Note that sometimes a join may be specified between a relation and itself. The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

$$((\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

The DIVISION Operation

The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications. An example is *Retrieve the names of employees who work on **all** the projects that 'John Smith' works on*. To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$SMITH \leftarrow \sigma_{Fname='John' \textbf{ AND } Lname='Smith'}(EMPLOYEE)$$

$$SMITH\_PNOS \leftarrow \pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn} SMITH)$$

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$SSN\_PNOS \leftarrow \pi_{Essn,\ Pno}(WORKS\_ON)$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$SSNS(Ssn) \leftarrow SSN\_PNOS \div SMITH\_PNOS$$

$$RESULT \leftarrow \pi_{Fname,\ Lname}(SSNS * EMPLOYEE)$$

The preceding operations are shown in Figure 8(a). The DIVISION operation can be expressed as a sequence of π, ×, and – operations as follows:

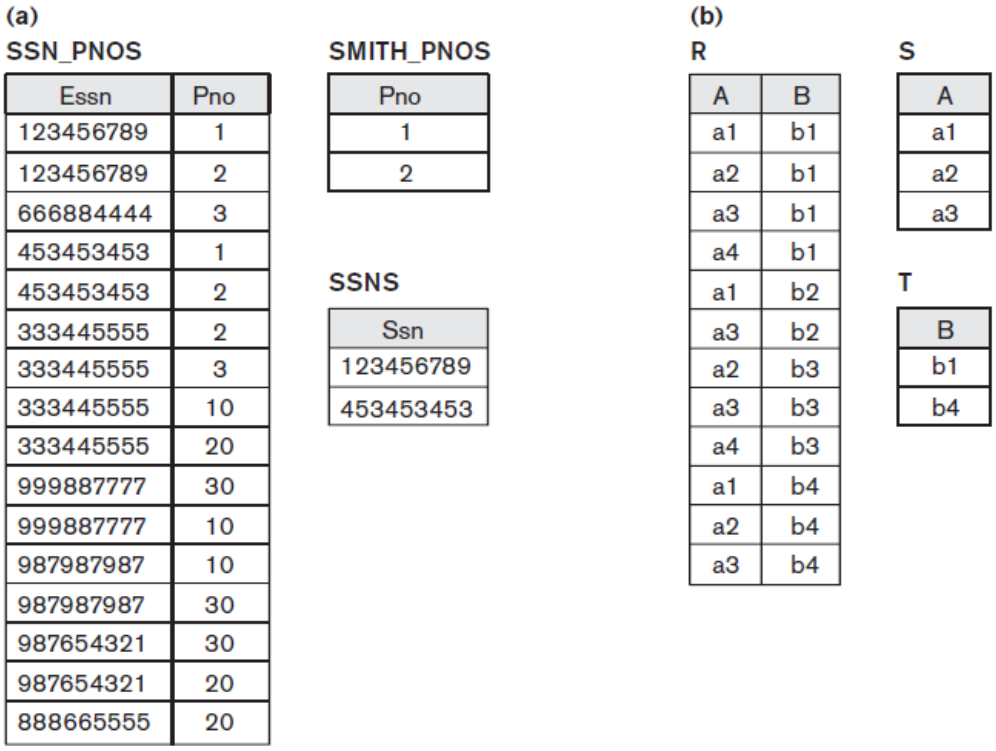
$$T1 \leftarrow \pi_{Y}(R)$$

$$T2 \leftarrow \pi_{Y}((S \times T1) - R)$$

$$T \leftarrow T1 - T2$$

The DIVISION operation is defined for convenience for dealing with queries that involve *universal quantification* or the *all* condition.

Figure 8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.



(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

OUTER JOIN Operations

The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation *R* * *S*, only tuples from *R* that have matching tuples in *S*—and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result. Tuples with NULL values in the join attributes are also eliminated. This type of join, where tuples with no match are eliminated, is known as an **inner join**. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in *R*, or all those in *S*, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values. For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*; if they do not manage

one, we can indicate it with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by ⟕, to retrieve the result as follows:

$$TEMP \leftarrow (EMPLOYEE \text{ ⟕}_{Ssn=Mgr\_ssn} DEPARTMENT)$$

$$RESULT \leftarrow \pi_{Fname,\ Minit,\ Lname,\ Dname}(TEMP)$$

The LEFT OUTER JOIN operation keeps every tuple in the *first*, or *left*, relation *R* in *R* ⟕ *S*; if no matching tuple is found in *S*, then the attributes of *S* in the join result are filled or *padded* with NULL values. The result of these operations is shown in Figure12.

A similar operation, **RIGHT OUTER JOIN**, denoted by ⟖, keeps every tuple in the *second*, or right, relation *S* in the result of *R*⟖*S*. A third operation, **FULL OUTER JOIN**, denoted by ⟗, keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

Fig.12. – Left outer join

The Tuple Relational Calculus

This section introduces the language known as **tuple relational calculus**, and another variation called **domain relational calculus**. In both variations of relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how, or *in what order*, to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request *in a particular order* of applying the operations; thus, it can be considered as a **procedural** way of stating a query.

A calculus expression may be written in different ways, but the way it is written has no bearing on how a query should be evaluated.

The relational calculus is important for two reasons. First, it has a firm basis in mathematical logic. Second, the standard query language (SQL) for RDBMSs has some of its foundations in the tuple relational calculus.

Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form:

{t | COND(t)}

where *t* is a tuple variable and COND(*t*) is a conditional (Boolean) expression involving *t* that evaluates to either TRUE or FALSE for different assignments of tuples to the variable *t*. The result of such a query is the set of all tuples *t* that evaluate COND(*t*) to TRUE. These tuples are said to **satisfy** COND(*t*). For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

{*t* | EMPLOYEE(*t*) **AND** *t*.Salary>50000}

The above query retrieves all attribute values for each selected EMPLOYEE tuple *t*. To retrieve only *some* of the attributes—say, the first and last names—we write

{*t*.Fname, *t*.Lname | EMPLOYEE(*t*) **AND** *t*.Salary>50000}

Informally, we need to specify the following information in a tuple relational calculus expression:

■ For each tuple variable *t*, the **range relation** *R* of *t*. This value is specified by a condition of the form *R*(*t*). If we do not specify a range relation, then the variable *t* will range over all possible tuples “in the universe” as it is not restricted to any one relation.

■ A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.

■ A set of attributes to be retrieved, the **requested attributes**. The values of these attributes are retrieved for each selected combination of tuples.

Let’s consider another query:

#. Retrieve the birth date and address of the employee (or employees) whose name is John B. Smith.

{*t*.Bdate, *t*.Address | EMPLOYEE(*t*) **AND** *t*.Fname='John' **AND** *t*.Minit='B' **AND** *t*.Lname='Smith'}

In tuple relational calculus, we first specify the requested attributes *t*.Bdate and *t*.Address for each selected tuple *t*. Then we specify the condition for selecting a tuple following the bar (|)—namely, that *t* be a tuple of the EMPLOYEE relation whose Fname, Minit, and Lname attribute values are 'John', 'B', and 'Smith', respectively.

Expressions and Formulas in Tuple Relational Calculus

A general **expression** of the tuple relational calculus is of the form

{t1.Aj, t2.Ak, ..., tn.Am | COND(t1, t2, ..., tn, tn+1, tn+2, ..., tn+m)}

where *t*1, *t*2, ..., *tn*, *tn*+1, ..., *tn*+*m* are tuple variables, each *Ai* is an attribute of the relation on which *ti* ranges, and COND is a **condition** or **formula** of the tuple relational calculus. A formula is made up of predicate calculus **atoms**, which can be one of the following:

1. An atom of the form *R*(*ti*), where *R* is a relation name and *ti* is a tuple variable. This atom identifies the range of the tuple variable *ti* as the relation whose name is *R*. It evaluates to TRUE if *ti* is a tuple in the relation *R*, and evaluates to FALSE otherwise.

2. An atom of the form *ti*.*A* **op** *tj*.*B*, where **op** is one of the comparison operators in the set {=, <, ≤, >, ≥, ≠}, *ti* and *tj* are tuple variables, *A* is an attribute of the relation on which *ti* ranges, and *B* is an attribute of the relation on which *tj* ranges.

3. An atom of the form *ti*.*A* **op** *c* or *c* **op** *tj*.*B*, where **op** is one of the comparison operators in the set {=, <, ≤, >, ≥, ≠}, *ti* and *tj* are tuple variables, *A* is an attribute of the relation on which *ti* ranges, *B* is an attribute of the relation on which *tj* ranges, and *c* is a constant value.

Each of the preceding atoms evaluates to either TRUE or FALSE for a specific combination of tuples; this is called the **truth value** of an atom. In general, a tuple variable *t* ranges over all possible tuples *in the universe*. For atoms of the form *R*(*t*), if *t* is assigned to a tuple that is a *member of the specified relation R*, the atom is TRUE; otherwise, it is FALSE. In atoms of types 2 and 3, if the tuple

variables are assigned to tuples such that the values of the specified attributes of the tuples satisfy the condition, then the atom is TRUE.

A **formula** (Boolean condition) is made up of one or more atoms connected via the logical operators **AND**, **OR**, and **NOT** and is defined recursively by Rules 1 and 2 as follows:

■ *Rule 1*: Every atom is a formula.

■ *Rule 2*: If *F*1 and *F*2 are formulas, then so are (*F*1 **AND** *F*2), (*F*1 **OR** *F*2), **NOT** (*F*1), and **NOT** (*F*2). The truth values of these formulas are derived from their component formulas *F*1 and *F*2 as follows:

a. (*F*1 **AND** *F*2) is TRUE if both *F*1 and *F*2 are TRUE; otherwise, it is FALSE.

b. (*F*1 **OR** *F*2) is FALSE if both *F*1 and *F*2 are FALSE; otherwise, it is TRUE.

c. **NOT** (*F*1) is TRUE if *F*1 is FALSE; it is FALSE if *F*1 is TRUE.

d. **NOT** (*F*2) is TRUE if *F*2 is FALSE; it is FALSE if *F*2 is TRUE.

The Existential and Universal Quantifiers

In addition, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** (∀) and the **existential quantifier** (∃). Truth values for formulas with quantifiers are described in Rules 3 and 4 below; first, however, we need to define the concepts of free and bound tuple variables in a formula.

Informally, a tuple variable *t* is bound if it is quantified, meaning that it appears in an (∃*t*) or (∀*t*) clause; otherwise, it is free. Formally, we define a tuple variable in a formula as **free** or **bound** according to the following rules:

- An occurrence of a tuple variable in a formula *F* that *is an atom* is free in *F*.
- An occurrence of a tuple variable *t* is free or bound in a formula made up of logical connectives—(*F*1 **AND** *F*2), (*F*1 **OR** *F*2), **NOT**(*F*1), and **NOT**(*F*2)— depending on whether it is free or bound in *F*1 or *F*2 (if it occurs in either). Notice that in a formula of the form *F* = (*F*1 **AND** *F*2) or *F* = (*F*1 **OR** *F*2), a tuple variable may be free in *F*1 and bound in *F*2, or vice

versa; in this case, one occurrence of the tuple variable is bound and the other is free in *F*.

- All *free* occurrences of a tuple variable *t* in *F* are **bound** in a formula *F*_ of the form *F*_= (∃*t*)(*F*) or *F*_ = (∀*t*)(*F*). The tuple variable is bound to the quantifier specified in *F*_. For example, consider the following formulas:

  *F*1 : *d*.Dname='Research'
  *F*2 : (∃*t*)(*d*.Dnumber=*t*.Dno)
  *F*3 : (∀*d*)(*d*.Mgr_ssn='333445555')

The tuple variable *d* is free in both *F*1 and *F*2, whereas it is bound to the (∀) quantifier in *F*3. Variable *t* is bound to the (∃) quantifier in *F*2.

We can now give Rules 3 and 4 for the definition of a formula we started earlier:

- Rule 3: If F is a formula, then so is (∃t)(F), where t is a tuple variable. The formula (∃t)(F) is TRUE if the formula F evaluates to TRUE for some (at least one) tuple assigned to free occurrences of t in F; otherwise, (∃t)(F) is FALSE.
- Rule 4: If F is a formula, then so is (∀t)(F), where t is a tuple variable. The formula (∀t)(F) is TRUE if the formula F evaluates to TRUE for every tuple (in the universe) assigned to free occurrences of t in F; otherwise, (∀t)(F) is FALSE.

The (∃) quantifier is called an existential quantifier because a formula (∃*t*)(*F*) is TRUE if *there exists* some tuple that makes *F* TRUE. For the universal quantifier, (∀*t*)(*F*) is TRUE if every possible tuple that can be assigned to free occurrences of *t* in *F* is substituted for *t*, and *F* is TRUE for *every such substitution*. It is called the universal or *for all* quantifier because every tuple in *the universe of* tuples must make *F* TRUE to make the quantified formula TRUE.

The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply, **domain calculus**. Historically, while SQL which was based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language called QBE (Query-By-Example), which is related

to domain calculus, was being developed almost concurrently at the IBM T.J. Watson Research Center in Yorktown Heights, New York. The formal specification of the domain calculus was proposed after the development of the QBE language and system.

Domain calculus differs from tuple calculus in the *type of variables* used in formulas: Rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree *n* for a query result, we must have *n* of these **domain variables**—one for each attribute. An expression of the domain calculus is of the form

{x1, x2, ..., xn | COND(x1, x2, ..., xn, xn+1, xn+2, ..., xn+m)}

where *x*1, *x*2, ..., *xn*, *xn*+1, *xn*+2, ..., *xn*+*m* are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

A formula is made up of **atoms**. The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form R(*x*1, *x*2, ..., *xj*), where *R* is the name of a relation of degree *j* and each *xi*, 1 ≤ *i* ≤ *j*, is a domain variable. This atom states that a list of values of <*x*1, *x*2, ..., *xj*> must be a tuple in the relation whose name is *R*, where *xi* is the value of the *i*th attribute value of the tuple. To make a domain calculus expression more concise, we can *drop the commas* in a list of variables; thus, we can write:
   a. {*x*1, *x*2, ..., *xn* | *R*(*x*1 *x*2 *x*3) AND ...}

   instead of:
   b. {*x*1, *x*2, ... , *xn* | *R*(*x*1, *x*2, *x*3) AND ...}
2. An atom of the form *xi* **op** *xj*, where **op** is one of the comparison operators in the set {=, <, ≤, >, ≥, ≠}, and *xi* and *xj* are domain variables.
3. An atom of the form *xi* **op** *c* or *c* **op** *xj*, where **op** is one of the comparison operators in the set {=, <, ≤, >, ≥, ≠}, *xi* and *xj* are domain variables, and *c* is a constant value.

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the truth values of the atoms. In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation

R, then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here. Some examples of queries specified in the domain calculus follow. We will use lowercase letters l, m, n, ..., x, y, z for domain variables.

Query 0. List the birth date and address of the employee whose name is 'John B. Smith'.

Q0: {u, v | (∃q) (∃r) (∃s) (∃t) (∃w) (∃x) (∃y) (∃z) (EMPLOYEE(qrstuvwxyz) AND q='John' AND r='B' AND s='Smith')}

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

Q1: {*q, s, v* | (∃*z*) (∃*l*) (∃*m*) (EMPLOYEE(*qrstuvwxyz*) **AND** DEPARTMENT(*lmno*) **AND** *l*='Research' **AND** *m=z*)}

SQL and INTEGRITY CONSTRAINTS

Basic SQL

The name SQL is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. A joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) has led to a standard version of SQL (ANSI 1986).

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL and a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++.

DDL (Data Definition Language)

- It is a set of SQL commands used to create, modify and delete database objects such as tables, views, indices, etc.
- It is normally used by DBA and database designers.
- It provides commands like:
  - ✓ CREATE: to create objects in a database.
  - ✓ ALTER: to alter the schema, or logical structure of the database.
  - ✓ DROP: to delete objects from the database.
  - ✓ TRUNCATE: to remove all records from the table.

The CREATE TABLE Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed.

Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing CREATE TABLE COMPANY.EMPLOYEE ... rather than CREATE TABLE EMPLOYEE ...

```
CREATE TABLE EMPLOYEE
        ( Fname          VARCHAR(15)      NOT NULL,
          Minit          CHAR,
          Lname          VARCHAR(15)      NOT NULL,
          Ssn            CHAR(9)          NOT NULL,
          Bdate          DATE,
          Address        VARCHAR(30),
          Sex            CHAR,
          Salary         DECIMAL(10,2),
          Super_ssn      CHAR(9),
          Dno            INT              NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE DEPARTMENT
        ( Dname          VARCHAR(15)      NOT NULL,
          Dnumber        INT              NOT NULL,
          Mgr_ssn        CHAR(9)          NOT NULL,
          Mgr_start_date DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );

CREATE TABLE DEPT_LOCATIONS
        ( Dnumber        INT              NOT NULL,
          Dlocation      VARCHAR(15)      NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE PROJECT
        ( Pname          VARCHAR(15)      NOT NULL,
          Pnumber        INT              NOT NULL,
          Plocation      VARCHAR(15),
          Dnum           INT              NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
```

```
CREATE TABLE WORKS_ON
        ( Essn              CHAR(9)          NOT NULL,
          Pno               INT              NOT NULL,
          Hours             DECIMAL(3,1)     NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );

CREATE TABLE DEPENDENT
        ( Essn              CHAR(9)          NOT NULL,
          Dependent_name    VARCHAR(15)      NOT NULL,
          Sex               CHAR,
          Bdate             DATE,
          Relationship      VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Attribute Data Types and Domains in SQL

The basic data types available for attributes include numeric, character string, bit string, Boolean, date, and time.

■ **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j)—where i, the precision, is the total number of decimal digits and j, the scale, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

■ **Character-string** data types are either fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters—or varying length—VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is case sensitive. For fixed length strings, a shorter string is padded with blank characters to the right. There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings in SQL. For example, 'abc' || 'XYZ' results in a single string 'abcXYZ'. Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

■ **Bit-string** data types are either of fixed length n—BIT(n)—or varying length—BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B‘10101’. Another variable-length bit string data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.

■ A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.

■ The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by the SQL implementation. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME; for example, DATE ‘2008-09-27’ or TIME ‘09:12:47’. In addition, a data type TIME(i), where i is called time fractional seconds precision, specifies i + 1 additional positions for TIME—one position for an additional period (.) separator character, and i positions for specifying decimal fractions of a second.

■ A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP ‘2008-09-27 09:12:47.648302’.

■ Another data type related to DATE, TIME, and TIMESTAMP is the **INTERVAL** data type. This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

It is possible to specify the data type of each attribute directly, alternatively, a domain can be declared, and the domain name used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

```
CREATE DOMAIN SSN_TYPE AS CHAR(9);
```

A domain can also have an optional default specification via a DEFAULT clause.

Specifying Constraints in SQL

Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. It is also possible to define a default value for an attribute by appending the clause DEFAULT <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute. If no default clause is specified, the default *default* value is NULL for attributes that do not have the NOT NULL constraint.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition.

```
Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);
```

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER
    CHECK (D_NUM > 0 AND D_NUM < 21);
```

Specifying Key and Referential Integrity Constraints

Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them. The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly.

```
Dnumber INT PRIMARY KEY;
```

The UNIQUE clause specifies alternate (secondary) keys. The UNIQUE clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:

```
Dname VARCHAR(15) UNIQUE;
```

Referential integrity is specified via the FOREIGN KEY clause. A referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to reject the update operation that will cause a violation, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken by attaching a referential triggered action clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE.

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing

attribute for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples.

Giving Names to Constraints

A constraint may be given a constraint name, following the keyword CONSTRAINT. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional.

Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other table constraints can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called tuple-based constraints because they apply to each tuple individually and are checked whenever a tuple is inserted or modified.

```
CHECK (Dept_create_date <= Mgr_start_date);
```

```
CREATE TABLE EMPLOYEE
    (  . . . ,
       Dno        INT             NOT NULL        DEFAULT 1,
    CONSTRAINT EMPPK
       PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
       FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                    ON DELETE SET NULL          ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
       FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                    ON DELETE SET DEFAULT       ON UPDATE CASCADE);
```

```
CREATE TABLE DEPARTMENT
    ( . . . ,
      Mgr_ssn    CHAR(9)       NOT NULL          DEFAULT '888665555',
      . . . ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                    ON DELETE SET DEFAULT   ON UPDATE CASCADE);

CREATE TABLE DEPT_LOCATIONS
    ( . . . ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE CASCADE              ON UPDATE CASCADE);
```

Basic Retrieval Queries in SQL

The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

SELECT <attribute list>
FROM <table list>
WHERE <condition>;

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively.

#1. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

SELECT Bdate, Address

```
FROM    EMPLOYEE
WHERE   Fname='John' AND Minit='B' AND Lname='Smith';
```

#2. Retrieve the name and address of all employees who work for the 'Research' department.

```
SELECT  Fname, Lname, Address
FROM    EMPLOYEE, DEPARTMENT
WHERE   Dname='Research' AND Dnumber=Dno;
```

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

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

Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is the case, and a multi table query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period.

```
SELECT  Fname, EMPLOYEE.Name, Address
FROM    EMPLOYEE, DEPARTMENT
WHERE   DEPARTMENT.Name='Research' AND
        DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. We can also create an *alias* for each table name to avoid repeated typing of long table names.

```
SELECT  EMPLOYEE.Fname, EMPLOYEE.LName, EMPLOYEE.Address
FROM    EMPLOYEE, DEPARTMENT
WHERE   DEPARTMENT.DName='Research' AND
        DEPARTMENT.Dnumber=EMPLOYEE.Dno;
```

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice, as in the following example.

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

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

In this case, we are required to declare alternative relation names E and S, called **aliases** or **tuple variables**, for the EMPLOYEE relation. An alias can follow the keyword **AS**, or it can directly follow the relation name. It is also possible to **rename** the relation attributes within the query in SQL by giving them aliases.

```
EMPLOYEE AS E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)
```

Unspecified WHERE Clause and Use of the Asterisk

A *missing* WHERE clause indicates no condition on tuple selection; hence, *all tuples* of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected.

```
# Select all EMPLOYEE Ssns
    SELECT    Ssn
    FROM      EMPLOYEE;

# Select all combinations of EMPLOYEE Ssn and DEPARTMENT Dname
    SELECT    Ssn, Dname
    FROM      EMPLOYEE, DEPARTMENT;
```

Selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not. It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result.

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes.*

```
#.  SELECT    *
    FROM      EMPLOYEE
    WHERE     Dno=5;
#.  SELECT    *
    FROM      EMPLOYEE, DEPARTMENT
    WHERE     Dname='Research' AND Dno=Dnumber;
#.  SELECT    *
    FROM      EMPLOYEE, DEPARTMENT;
```

Tables as Sets in SQL

SQL usually treats a table not as a set but rather as a **multiset**; *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- ✓ Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- ✓ The user may want to see duplicate tuples in the result of a query.
- ✓ When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not.

#. Retrieve the salary of every employee.

```
SELECT    ALL Salary
FROM      EMPLOYEE;
```

#. Retrieve all distinct salary values.

```
SELECT    DISTINCT Salary
FROM      EMPLOYEE;
```

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra. There are set union (**UNION**), set difference (**EXCEPT**), and set intersection (**INTERSECT**) operations.

The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result.* These set operations apply only to *union compatible relations*, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations.

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

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

Substring Pattern Matching and Arithmetic Operators

Now we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the **LIKE** comparison operator. This

can be used for string **pattern matching**. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character.

#. Retrieve all employees whose address is in Houston, Texas.

```
SELECT    Fname, Lname
FROM      EMPLOYEE
WHERE     Address LIKE '%Houston,TX%';
```

#. Find all employees who were born during the 1950s.

```
SELECT    Fname, Lname
FROM      EMPLOYEE
WHERE     Bdate LIKE '_ _ 5 _ _ _ _ _ _ _ _';
```

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character*, which is specified after the string using the keyword ESCAPE. For example, 'AB_CD\%EF' ESCAPE '\' represents the literal string 'AB_CD%EF' because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (' ') if they are to be included in a string because they are used to begin and end strings. If an apostrophe (') is needed, it is represented as two consecutive apostrophes (") so that it will not be interpreted as ending the string. Notice that substring comparison implies that attribute values are not atomic (indivisible) values, as we had assumed in the formal relational model.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains.

```
SELECT    E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
FROM      EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE     E.Ssn=W.Essn AND W.Pno=P.Pnumber AND
          P.Pname='ProductX';
```

For string data types, the concatenate operator || can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (–) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator, which can be used for convenience, is **BETWEEN**.

#. Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
SELECT    *
FROM      EMPLOYEE
```

```
WHERE    (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause.

#. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
SELECT    D.Dname, E.Lname, E.Fname, P.Pname
FROM      DEPARTMENT D, EMPLOYEE E, WORKS_ON W, PROJECT P
WHERE     D.Dnumber= E.Dno AND E.Ssn= W.Essn AND
          W.Pno = P.Pnumber
ORDER BY  D.Dname, E.Lname, E.Fname;
```

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly.

INSERT, DELETE, and UPDATE Statements in SQL

The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command.

```
INSERT INTO    EMPLOYEE
VALUES         ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak
               Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification *and* no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be *left out.*

```
INSERT INTO    EMPLOYEE (Fname, Lname, Dno, Ssn)
VALUES         ('Richard', 'Marini', 4, '653298653');
```

Attributes not specified are set to their DEFAULT or to NULL, and the values are listed in the same order as the *attributes are listed in the INSERT* command itself. It is also possible to insert into a relation *multiple tuples* separated by commas in a single

INSERT command. The attribute values forming *each tuple* are enclosed in parentheses.

A DBMS that fully implements SQL should support and enforce all the integrity constraints that can be specified in the DDL. A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query.*

```
CREATE TABLE    WORKS_ON_INFO
( Emp_name      VARCHAR(15),
Proj_name       VARCHAR(15),
Hours_per_week  DECIMAL(3,1) );

INSERT INTO     WORKS_ON_INFO ( Emp_name, Proj_name,
                Hours_per_week )
SELECT          E.Lname, P.Pname, W.Hours
FROM            PROJECT P, WORKS_ON W, EMPLOYEE E
WHERE           P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition.

```
DELETE FROM     EMPLOYEE
WHERE           Lname='Brown';

DELETE FROM     EMPLOYEE
WHERE           Ssn='123456789';

DELETE FROM     EMPLOYEE
WHERE           Dno=5;

DELETE FROM     EMPLOYEE;
```

The UPDATE Command

The **UPDATE** command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a

referential triggered action is specified in the referential integrity constraints of the DDL. An additional **SET** clause in the UPDATE command specifies the attributes to be modified and their new values.

```
UPDATE    PROJECT
SET       Plocation = 'Bellaire', Dnum = 5
WHERE     Pnumber=10;
```

Several tuples can be modified with a single UPDATE command.

```
UPDATE    EMPLOYEE
SET       Salary = Salary * 1.1
WHERE     Dno = 5;
```

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or **IS NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN).

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

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

Nested Queries, Tuples, and Set / Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**. Below query introduces the comparison operator **IN**, which compares a value *v* with a set (or multiset) of values *V* and evaluates to **TRUE** if *v* is one of the elements in *V*.

The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, while the second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

```
SELECT DISTINCT     Pnumber
```

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

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

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

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on. In this example, the IN operator compares the sub tuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value *v* (typically an attribute name) to a set or multiset *v* (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value *v* is equal to *some value* in the set *V* and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (*v* > ALL *V*) returns TRUE if the value *v* is greater than *all* the values in the set (or multiset) *V*.

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

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

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

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

In this nested query, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not *have to* qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fname and Ssn, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities.

Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**. We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query.* For example, we can think of previous query as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, query may be written as:

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

The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is *empty* (contains no tuples) or not. The result of EXISTS is a Boolean value

TRUE if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples.

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

#. Retrieve the names of employees who have no dependents.

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

#. List the names of managers who have at least one dependent.

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

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

```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     NOT EXISTS ( SELECT   *
                       FROM     WORKS_ON B
```

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

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

Explicit Sets and Renaming of Attributes in SQL

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

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

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

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses.

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

Joined Tables in SQL and Outer Joins

The concept of a joined table (or joined relation) permit users to specify a table resulting from a join operation in the FROM clause of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause.

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

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

The FROM clause in query contains a single joined table. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN.

```
SELECT   Fname, Lname, Address
FROM     (EMPLOYEE NATURAL JOIN
         (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
WHERE    Dname='Research';
```

The default type of join in a joined table is called an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation.

If the user requires all tuples be included, an OUTER JOIN must be used explicitly. In SQL, this is handled by explicitly specifying the keyword OUTER JOIN in a joined table.

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

It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a multiway join.

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

Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. Grouping is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications.

A number of built-in aggregate functions exist: COUNT, SUM, MAX, MIN, and AVG. The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause. The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another.

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

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

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

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

#. Retrieve the total number of employees in the company.

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

#. Retrieve total Number of employees in the 'Research' department.

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

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

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

#. Retrieve the names of all employees who have two or more dependents.

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

The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees in each department or the number of employees who work on each project. SQL has a GROUP BY clause for this purpose.

The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

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

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

```
FROM        EMPLOYEE
GROUP BY    Dno;
```

If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute.

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

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

SQL also provides a HAVING clause, which can appear in conjunction with a GROUP BY clause. HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

#. For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

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

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

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

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

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

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

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

Constraints as Assertions

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

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

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT     *
                     FROM       EMPLOYEE E, EMPLOYEE M,
                                DEPARTMENT D
                     WHERE      E.Salary>M.Salary
                                AND E.Dno=D.Dnumber
                                AND D.Mgr_ssn=M.Ssn ) );
```

The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated.

The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition. By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty.

A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL only when tuples are inserted or updated.

Hence, constraint checking can be implemented more efficiently by the DBMS in these cases.

Triggers

In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. The CREATE TRIGGER statement is used to implement such actions in SQL.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database. Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION, which will notify the supervisor. The trigger could then be written as:

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
        ON EMPLOYEE
FOR EACH ROW
        WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
                              WHERE SSN = NEW.SUPERVISOR_SSN ) )
                             INFORM_SUPERVISOR(NEW.Supervisor_ssn,
                             NEW.Ssn );
```

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger has three components:

1. The event(s): These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. These events are specified after the keyword BEFORE in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword AFTER, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The condition that determines whether the rule action should be executed: Once the triggering event has occurred, an optional condition may be evaluated. If no condition is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

3. The action to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically

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

Views (Virtual Tables) in SQL

A view in SQL terminology is a single table that is derived from other tables. These other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

In SQL, the command to specify a view is CREATE VIEW. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

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

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

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as:

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

A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized or materialized at the time of view definition but rather at the time when we specify a query on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date. If we do not need a view any more, we can use the DROP VIEW command to dispose of it.

```
DROP VIEW    WORKS_ON1;
```

View Implementation, View Update, and Inline Views

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables. For example, the previous query would be automatically modified to the following query by the DBMS:

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

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are going to be applied to the same view within a short period of time. The second strategy, called **view materialization**, involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of incremental update have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a materialized view table when a database update is applied to one of the defining base tables. The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base

relations in multiple ways. Hence, it is often not possible for the DBMS to determine which of the updates is intended. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'.

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

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create additional side effects that affect the result of other queries. For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation:

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

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

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the user who specified the view update wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

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

```
UPDATE    DEPT_INFO
SET       Total_sal=100000
```

```
WHERE       Dname='Research';
```

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

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

In summary, we can make the following observations:

■ A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified.

■ Views defined on multiple tables using joins are generally not updatable.

■ Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause WITH CHECK OPTION must be added at the end of the view definition if a view is to be updated. This allows the system to check for view updatability

and to plan an execution strategy for view updates. It is also possible to define a view table in the FROM clause of an SQL query. This is known as an in-line view. In this case, the view is defined within the query itself.

Schema Change Statements in SQL

The DROP Command

The DROP command can be used to drop named schema elements, such as tables, domains, or constraints. One can also drop a schema. There are two drop behavior options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

```
DROP SCHEMA COMPANY CASCADE;
```

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

If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command.

```
DROP TABLE DEPENDENT CASCADE;
```

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints. The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

The ALTER Command

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

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

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

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

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

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

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

One can also change the constraints specified on a table by adding or dropping a named constraint. To be dropped, a constraint must have been given a name when it was specified.

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

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.

Database Stored Procedures

It is sometimes useful to create database program modules—procedures or functions—that are stored and executed by the DBMS at the database server. These are historically known as database stored procedures, although they can be functions or procedures. The term used in the SQL standard for stored procedures is persistent stored modules because these programs are stored persistently by the DBMS, similarly to the persistent data stored by the DBMS.

Stored procedures are useful in the following circumstances:

■ If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.

■ Executing a program at the server can reduce data transfer and communication cost between the client and server in certain situations.

■ These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users. Additionally, they can be used to check for complex constraints that are beyond the specification power of assertions and triggers.

In general, many commercial DBMSs allow stored procedures and functions to be written in a general-purpose programming language. Alternatively, a stored procedure can be made of simple SQL commands such as retrievals and updates. The general form of declaring stored procedures is as follows:

CREATE PROCEDURE *<procedure name> (<parameters>)*
<local declarations>
<procedure body> ;

The parameters and local declarations are optional, and are specified only if needed. For declaring a function, a return type is necessary, so the declaration form is

CREATE FUNCTION *<function name> (<parameters>)*
RETURNS <return type>
<local declarations>
<function body> ;

The CALL statement in the SQL standard can be used to invoke a stored procedure—either from an interactive interface or from embedded SQL or SQLJ.

CALL *<procedure or function name> (<argument list>) ;*

Extending SQL for Specifying Persistent Stored Modules

The conditional branching statement in SQL/PSM has the following form:

IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;

```
//Function PSM1:
CREATE FUNCTION Dept_size(IN deptno INTEGER)
   RETURNS VARCHAR [7]
DECLARE No_of_emps INTEGER ;
SELECT  COUNT(*) INTO No_of_emps
   FROM   EMPLOYEE WHERE Dno = deptno ;
IF No_of_emps > 100 THEN RETURN "HUGE"
        ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
        ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
        ELSE RETURN "SMALL"
END IF ;
```

SQL/PSM has several constructs for looping. There are standard while and repeat looping structures, which have the following forms:

WHILE <condition> DO
<statement list>
END WHILE ;

REPEAT
<statement list>
UNTIL <condition>
END REPEAT ;

There is also a cursor-based looping structure. The statement list in such a loop is executed once for each tuple in the query result. This has the following form:

FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
<statement list>
END FOR ;

Table Summary of SQL Syntax

```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                            { , <column name> <column type> [ <attribute constraint> ] }
                            [ <table constraint> { , <table constraint> } ] )

DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>

SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]

<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                       { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )

<grouping attributes> ::= <column name> { , <column name> }

<order> ::= ( ASC | DESC )

INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )

DELETE FROM <table name>
[ WHERE <selection condition> ]

UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[ WHERE <selection condition> ]

CREATE [ UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Stored Procedure

A stored procedure (proc) is a group of PL/SQL statements that performs specific task. A procedure has two parts, header and body. The header consists of the name of the procedure and the parameters passed to the procedure. The body consists of declaration section, execution section. A procedure may or may not return any value. A procedure may return more than one value.

General Syntax to create or Alter a procedure

CREATE [OR ALTER] PROCEDURE proc_name [list of parameters]
Declaration section
AS
BEGIN
Execution section
END;

We can pass parameters to the procedures in three ways.

IN-parameters: - These types of parameters are used to send values to stored procedures.

OUT-parameters: - These types of parameters are used to get values from stored procedures. This is similar to a return type in functions but procedure can return values for more than one parameters.

IN OUT-parameters: - This type of parameter allows us to pass values into a procedure and get output values from the procedure.

There are two ways to execute a procedure.

1) From the SQL prompt.
   Syntax: *EXECUTE [or EXEC] procedure_name (parameter);*
2) Within another procedure – simply use the procedure name.
   Syntax: *procedure_name (parameter);*

UNIT 3 – RELATIONAL DATABASE DESIGN

Relational Database Design:

Functional Dependency, Different anomalies in designing a Database., Normalization using functional dependencies, Decomposition, Boyce-Codd Normal Form, 3NF, Normalization using multi-valued dependencies, 4NF, 5NF.

Internals of RDBMS: Physical data structures, Query optimization: join algorithm, statistics and cost base optimization. Transaction processing, Concurrency control and Recovery Management: transaction model properties, state serializability, lock base protocols, two phase locking.

Functional Dependency

A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples $t1$ and $t2$ in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) determine the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X. The abbreviation for functional dependency is **FD** or **f.d.** The set of attributes X is called the left-hand side of the FD, and Y is called the **right-hand side**. The left hand side of the FD is also referred as **determinant** whereas the right hand side of the FD is referred as **dependent**.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of r(R) agree on their X-value, they must necessarily agree on their Y value. Note the following:

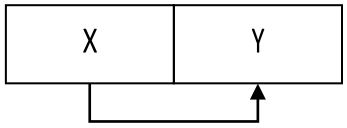
■ If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance r(R)—that is, X is a candidate key of R—this implies that $X \rightarrow Y$ for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state r(R) will have the same value of X). If X is a candidate key of R, then $X \rightarrow R$.

■ If X→Y in R, this does not say whether or not Y→X in R.

Consider the relation schema EMP_PROJ in Figure (b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a. Ssn→Ename
b. Pnumber →{Pname, Plocation}
c. {Ssn, Pnumber}→Hours

Diagrammatic Representation

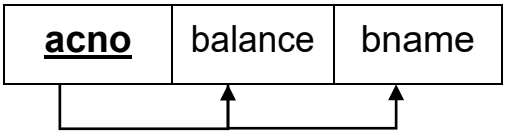


R:

Consider the relation Account (acno, balance, bname). In this relation acno can determines balance and bname. So, there is a functional dependency from acno to balance and bname. This can be denoted by

acno → {balance, bname}.

Account:



Different anomalies in designing a Database

Anomalies are problems that can occur in poorly planned, un-normalized database where all the data are stored in one table.

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT defined earlier with that for an EMP_DEPT base relation in Figure, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values

pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department. In contrast, each department's information appears only once in the DEPARTMENT relation. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation. Storing natural joins of base relations leads to an additional problem referred to as update anomalies. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.

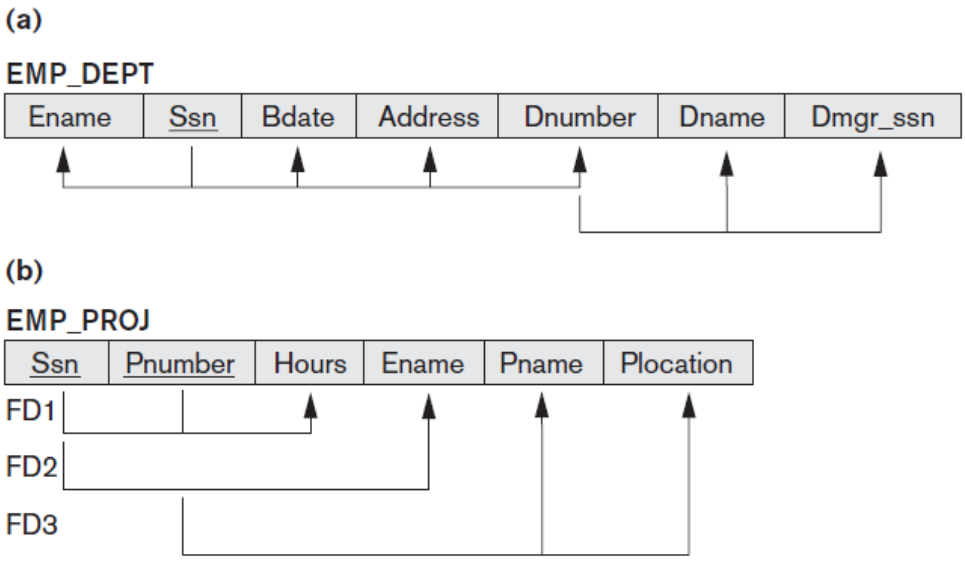


Figure
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.

Insertion Anomalies

Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

■ To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are consistent with the corresponding values for department 5 in other tuples in EMP_DEPT. In the previous design of we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation.

■ It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because Ssn is its

primary key. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values any more. This problem does not occur in the previous design because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies

If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in the earlier database because DEPARTMENT tuples are stored separately.

Modification Anomalies

In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.

Decomposition

Decomposition is the process of breaking down given relation into two or more relations. Here, relation R is replaced by two or more relations in such a way that:

1. Each new relation contains a subset of the attributes of R, and
2. Together, they all include all tuples and attributes of R.

Relational database design process starts with a universal relation schema R = {A1, A2, A3,..., An), which includes all the attributes of the database. The universal relation states that every attribute name is unique.

Using functional dependencies, this universal relation schema is decomposed into a set of relation schemas D = {R1, R2, R3,…,Rm}. Now, D becomes the relational database schema and D is referred as decomposition of R. Generally, decomposition is used to eliminate the pitfalls of the poor database design during normalization process.

For example, consider the relation Account_Branch given in figure:

Account_Branch			
Ano	**Balance**	**Bname**	**Baddress**
A01	5000	Vvn	Mota bazaar,VVNagar

A02	6000	Ksad	Chhota bazaar, Karamsad
A03	7000	Anand	Nana bazaar, Anand
A04	8000	Ksad	Chhota bazaar, Karamsad
A05	6000	Vvn	Mota bazaar,VVNagar

This relation can be divided with two different relations

1. Account (Ano, Balance, Bname)
2. Branch (Bname, Baddress)

Account		
Ano	**Balance**	**Bname**
A01	5000	Vvn
A02	6000	Ksad
A03	7000	Anand
A04	8000	Ksad
A05	6000	Vvn

Branch	
Bname	**Baddress**
Vvn	Mota bazaar, VVNagar
Ksad	Chhota bazaar, Karamsad
Anand	Nana Bazar,Anand

A decomposition of relation can be either lossy decomposition or lossless decomposition. There are two types of decomposition:

1. lossy decomposition
2. lossless decomposition (non-loss decomposition)

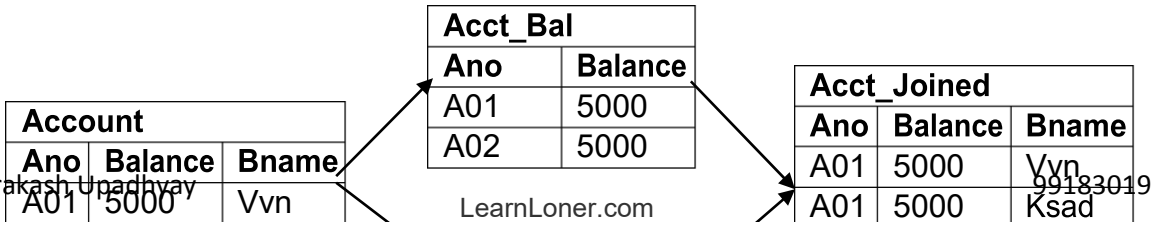
Lossy Decomposition

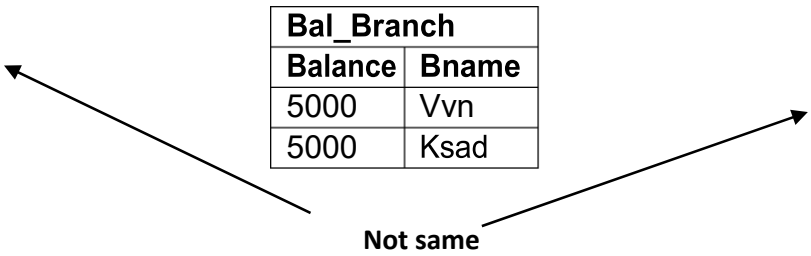
The decomposition of relation R into R1 and R2 is lossy when the join of R1 and R2 does not yield the same relation as in R. The disadvantage of such kind of decomposition is that some information is lost during retrieval of original relation. And so, such kind of decomposition is referred as lossy decomposition. From practical point of view, decomposition should not be lossy decomposition.

Example

A figure shows a relation Account. This relation is decomposed into two relations Acc_Bal and Bal_Branch. Now, when these two relations are joined on the common attribute Balance, the resultant relation will look like Acct_Joined. This Acct_Joined relation contains rows in addition to those in original relation Account. Here, it is not possible to specify that in which branch account A01 or A02 belongs.

So, information has been lost by this decomposition and then join operation.





In other words, decomposition is lossy if decompose into R1 and R2 and again combine (join) R1 and R2 we cannot get original table as R1, over X, where R is an original relation, R1 and R2 are decomposed relations, and X is a common attribute between these two relations.

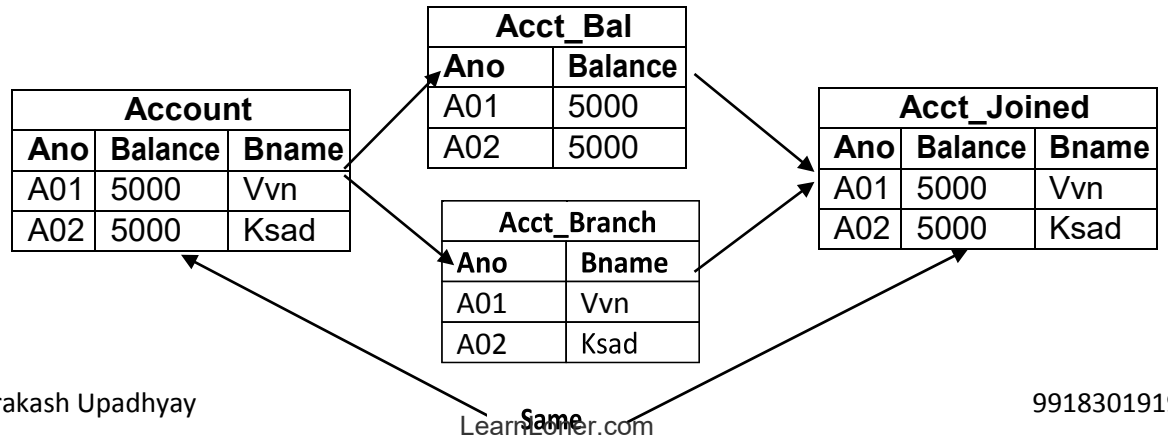
Lossless (Non-loss) Decomposition

The decomposition of relation R into R1 and R2 is lossless when the join of R1 and R2 produces the same relation as in R. This is also referred as a non-additive (non-loss) decomposition. All decompositions must be lossless.

Example

Again, the same relation Account is decomposed into two relations Acct_Bal and Acct_Branch. Now, when these two relations are joined on the common column Ano, the resultant relation will look like Acc_Joined relation. This relation is exactly same as that of original relation Account. In other words, all the information of original relation is preserved here. In lossless decomposition, no any fake tuples are generated when a natural join is applied to the relations in the decomposition.

In other words, decomposition is lossy if R = join of R1 and R2, over X, where R is an original relation, R1 an R2 are decomposed relations, and x is a common attribute between these two relations.



Normalization using Functional Dependencies

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**. The process, proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary. Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies. It can be considered as a "filtering" or "purification" process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with the following:

■ A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes.

■ A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree.

First Normal Form

It states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. The only attribute values permitted by 1NF are single atomic (or indivisible) values.

Consider the DEPARTMENT relation schema shown in Figure, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure (a). We assume that each department can have a number of locations. The DEPARTMENT schema and a sample relation state are shown in Figure. As we can

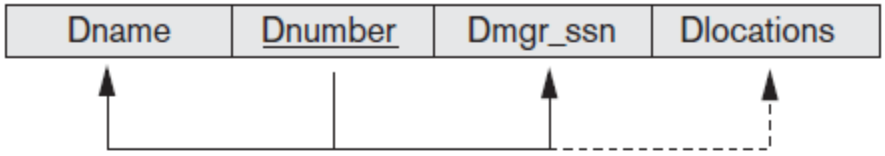
see, this is not in 1NF because Dlocations is not an atomic attribute. There are two ways we can look at the Dlocations attribute:

■ The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.

■ The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber→Dlocations because each set is considered a single member of the attribute domain.

(a)

DEPARTMENT



(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

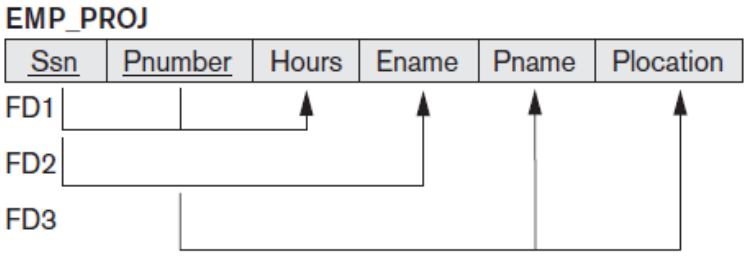
In either case, the DEPARTMENT relation in Figure 15.9 is not in 1NF; There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation}, as shown in Figure. A distinct tuple in DEPT_LOCATIONS exists for each location of a department. This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure (c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing redundancy in the relation.
3. If a maximum number of values is known for the attribute—for example, if it is known that at most three locations can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing NULL values if most departments have fewer than three locations. Querying on this attribute becomes more difficult; for example, consider how you would write the query: List the departments that have 'Bellaire' as one of their locations in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general.

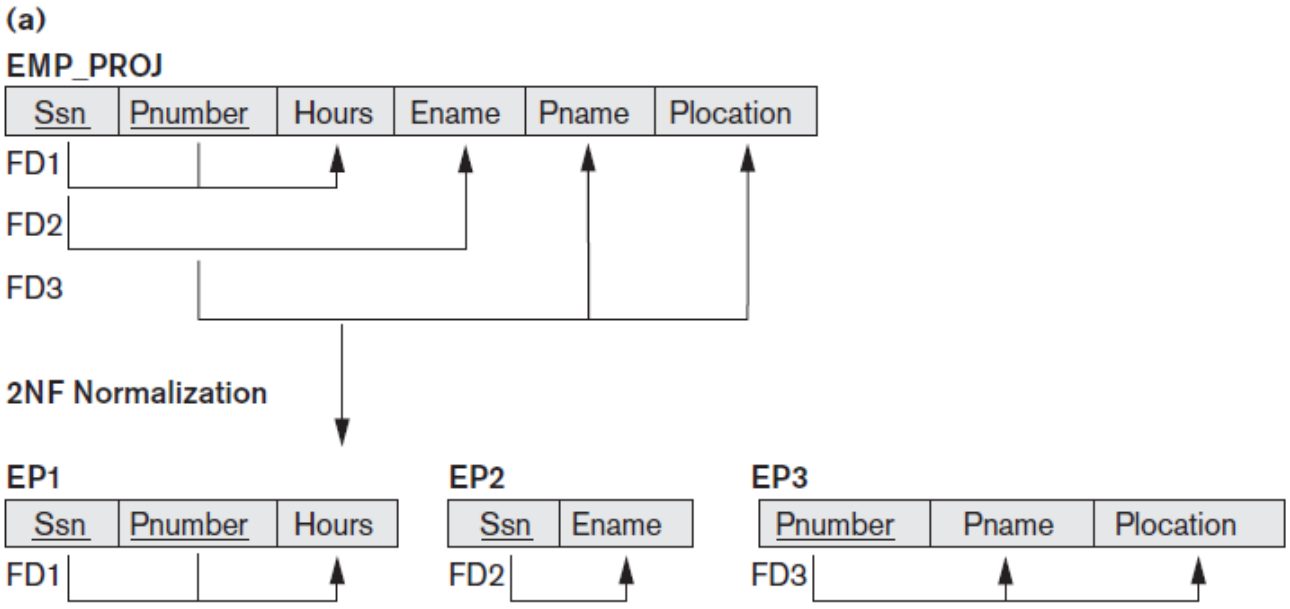
Second Normal Form

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency X → Y is a full functional dependency if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute A ε X, (X – {A}) does not functionally determine Y. A functional dependency X→Y is a partial dependency if some attribute A ε X can be removed from X and the dependency still holds; that is, for some A ε X, (X – {A}) → Y. In Figure, {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber→Hours holds). However, the dependency {Ssn, Pnumber}→Ename is partial because Ssn→Ename holds.



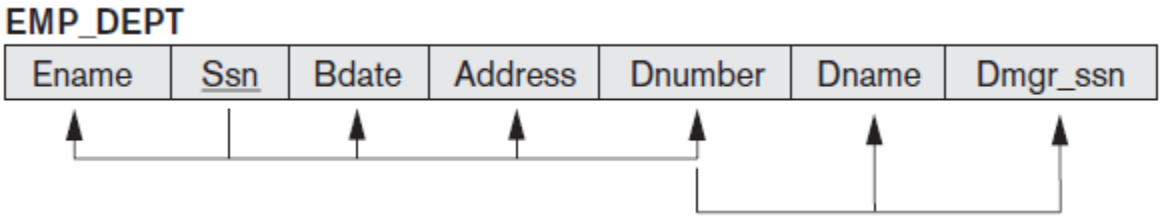
Definition. *A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.*

If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure above lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure (a) below, each of which is in 2NF.



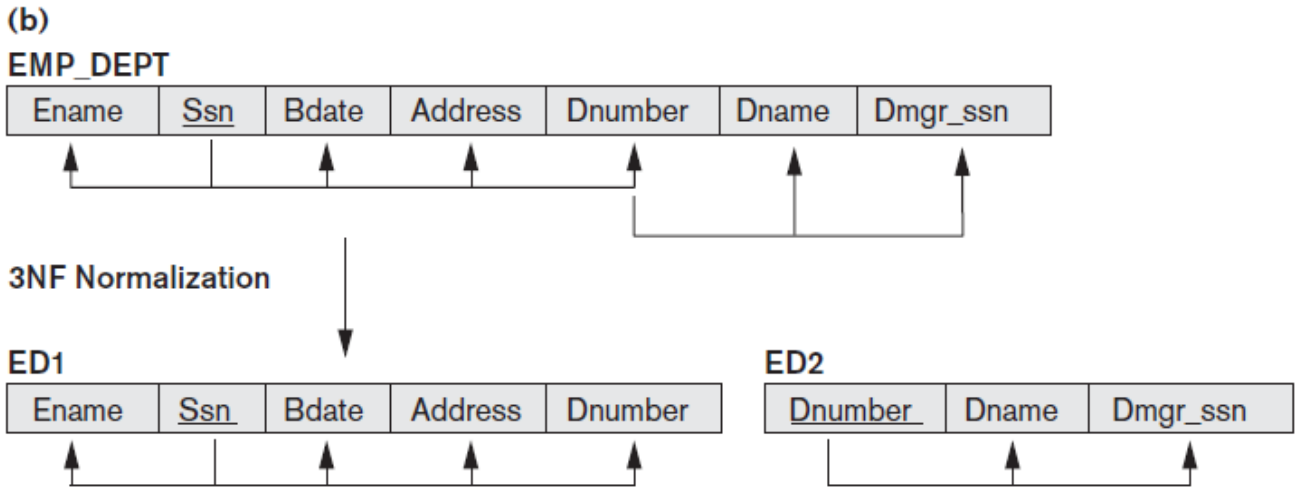
Third Normal Form

Third normal form (3NF) is based on the concept of transitive dependency. A functional dependency X→Y in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both X→Z and Z→Y hold. The dependency Ssn→Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure below, because both the dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn hold and Dnumber is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of Dmgr_ssn on Dnumber is undesirable in EMP_DEPT since Dnumber is not a key of EMP_DEPT.



Definition. *According to Codd's original definition, a relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key.*

The relation schema EMP_DEPT in Figure is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure (b) below. Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.



General Definition of Second Normal Form

Definition. A relation schema *R* is in **second normal form (2NF)** if every nonprime attribute *A* in *R* is not partially dependent on *any* key of *R*.

Consider the relation schema LOTS shown in Figure, which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state. Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 in Figure (a) hold. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate

FD4: Area → Price

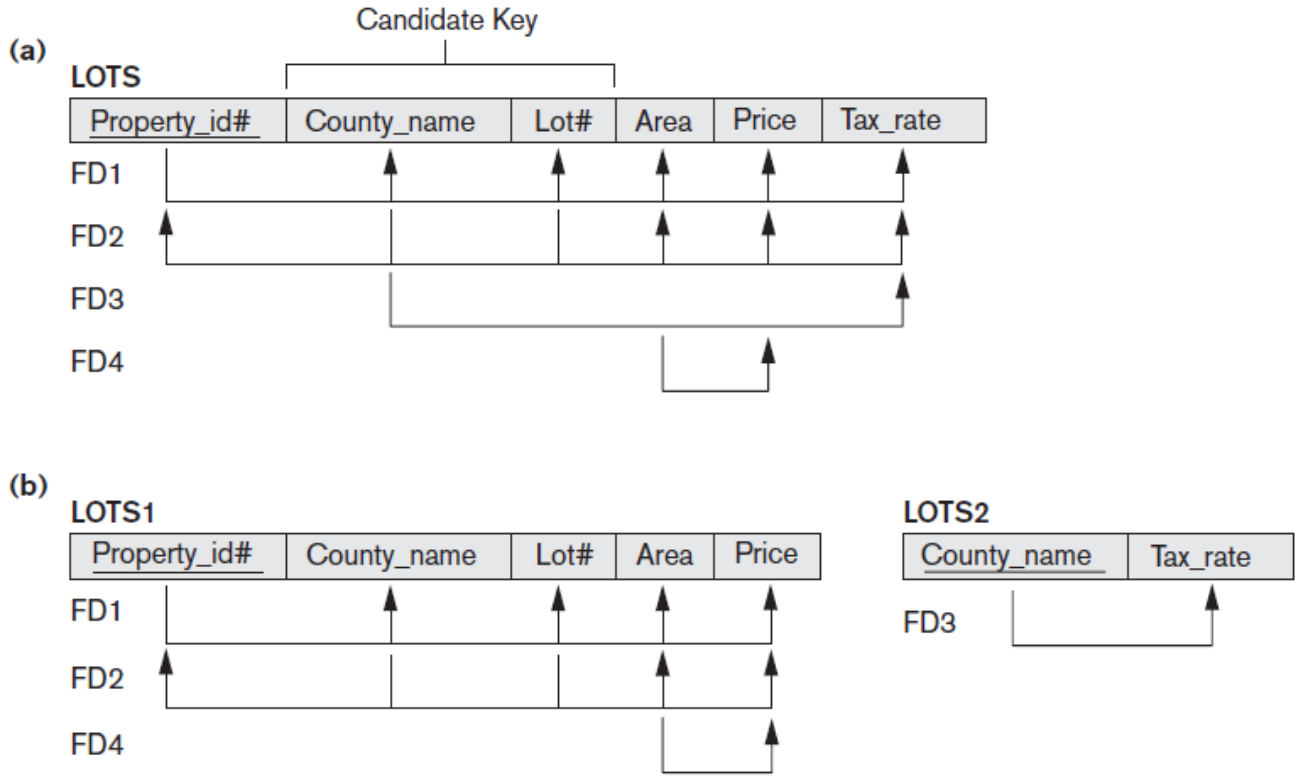
In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.) The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name,

Lot#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure (b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF.

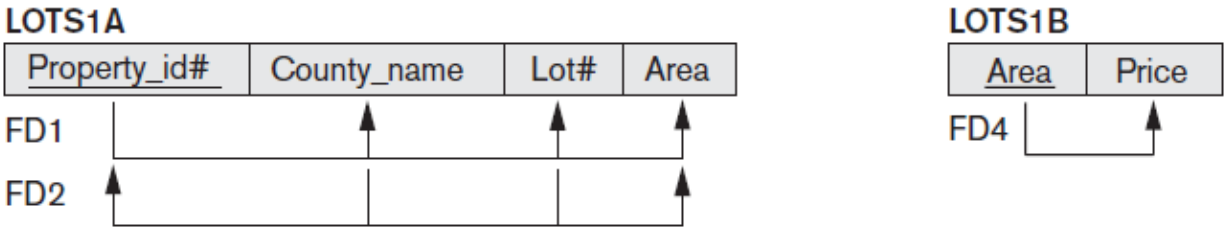
General Definition of Third Normal Form

Definition. A relation schema *R* is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in *R*, either (a) *X* is a superkey of *R*, or (b) *A* is a prime attribute of *R*.

According to this definition, LOTS2 is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B. We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the left hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.



(c)



A relation schema R violates the general definition of 3NF if a functional dependency X → A holds in R that does not meet either condition—meaning that it violates both conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

■ A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.

■ A proper subset of a key of R functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a general alternative definition of 3NF as follows:

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

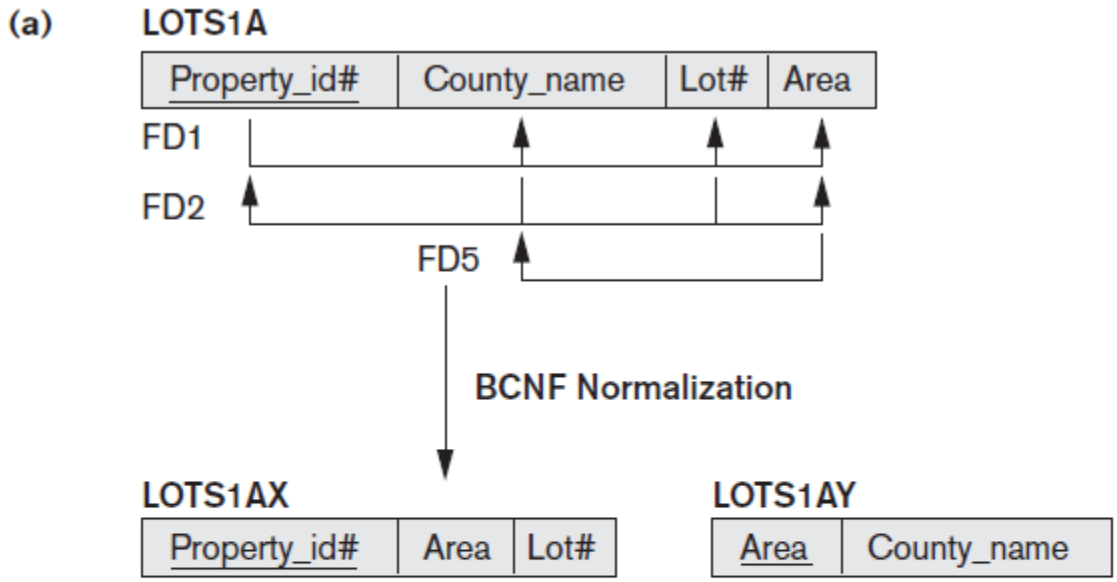
■ It is fully functionally dependent on every key of R.

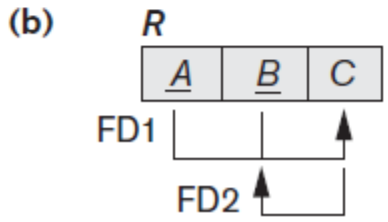
■ It is nontransitively dependent on every key of R.

Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Consider again the LOTS relation schema in Figure (a) with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: Area→County_name. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a prime attribute. The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation *R*(Area, County_name), since there are only 16 possible Area values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

Definition. A relation schema *R* is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in *R*, then *X* is a superkey of *R*.





As another example, consider Figure, which shows a relation TEACH with the following dependencies:

FD1: {Student, Course} → Instructor

FD2: Instructor → Course

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure (b), with Student as A, Course as B, and Instructor as C. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. *{Student, Instructor} and {Student, Course}.*
2. *{Course, Instructor} and {Course, Student}.*
3. *{Instructor, Course} and {Instructor, Student}.*

All three decompositions lose the functional dependency FD1. The desirable decomposition of those just shown is 3 because it will not generate spurious tuples after a join.

Multivalued Dependency

Fourth Normal Form

Apart from functional dependency, in many cases relations have constraints that cannot be specified as functional dependencies. These constraints are defined using multivalued dependency (MVD) and fourth normal is based on this dependency.

For example, consider the relation EMP shown in Figure (a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another. To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a

multivalued dependency on the EMP relation. Informally, whenever two independent 1:N relationships A:B and A:C are mixed in the same relation, R(A, B, C), an MVD may arise.

(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

Definition. A multivalued dependency X→→Y specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t1 and t2 exist in r such that t1[X] = t2[X], then two tuples t3 and t4 should also exist in r with the following properties, where we use Z to denote (R – (X ∪ Y)):

- *t3[X] = t4[X] = t1[X] = t2[X].*
- *t3[Y] = t1[Y] and t4[Y] = t2[Y].*
- *t3[Z] = t2[Z] and t4[Z] = t1[Z].*

Whenever X→→Y holds, we say that X multidetermines Y. Because of the symmetry in the definition, whenever X →→ Y holds in R, so does X →→ Z. Hence, X →→ Y implies X→→Z, and therefore it is sometimes written as X→→Y|Z. An MVD X →→ Y in R is called a trivial(of little value or importance) MVD if (a) Y is a subset of X, or (b) X ∪ Y = R. For example, the relation EMP_PROJECTS in Figure (b) has the trivial MVD Ename →→ Pname. An MVD that satisfies neither (a) nor (b) is called a nontrivial MVD. A trivial MVD will hold in any relation state r of R; it is called trivial because it does not specify any significant or meaningful constraint on R.

If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. This redundancy is clearly undesirable. However, the EMP schema is in BCNF because no functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP.

Definition. A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every nontrivial multivalued dependency X →→ Y in F, X is a superkey for R.

We can state the following points:

- An all-key relation is always in BCNF since it has no FDs.

■ An all-key relation such as the EMP relation in Figure (a), which has no FDs but has the MVD Ename→→ Pname | Dname, is not in 4NF.

■ A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.

■ The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure (a). EMP is not in 4NF because in the nontrivial MVDs Ename→→ Pname and Ename→→ Dname, and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure (b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename →→ Pname in EMP_PROJECTS and Ename →→ Dname in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

Join Dependencies and Fifth Normal Form

Definition. A join dependency (JD), denoted by JD(R1, R2, ..., Rn), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R1, R2, ..., Rn. Hence, for every such r we have

$$* (\pi R1(r), \pi R2(r), ..., \pi Rn(r)) = r$$

Notice that an MVD is a special case of a JD where n = 2. That is, a JD denoted as JD(R1, R2) implies an MVD (R1 ∩ R2) →→ (R1 – R2) (or, by symmetry, (R1 ∩ R2)→→(R2 – R1)). A join dependency JD(R1, R2, ..., Rn), specified on relation schema R, is a trivial JD if one of the relation schemas Ri in JD(R1, R2, ..., Rn) is equal to R. Such a dependency is called trivial because it has the nonadditive join property for any relation state r of R and thus does not specify any constraint on R.We can now define fifth normal form.

Definition. A relation schema R is in fifth normal form (5NF) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R1, R2, ..., Rn) in F+ (that is, implied by F), every Ri is a superkey of R.

(c) **SUPPLY**

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) R_1

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

For an example of a JD, consider the SUPPLY all-key relation in Figure (c). Suppose that the following additional constraint always holds: Whenever a supplier s supplies part p, and a project j uses part p, and the supplier s supplies at least one part to project j, then supplier s will also be supplying part p to project j. This constraint can be restated in other ways and specifies a join dependency JD(R1,R2, R3) among the three projections R1(Sname, Part_name), R2(Sname, Proj_name), and R3(Part_name, Proj_name) of SUPPLY. If this constraint holds, the tuples below the dashed line in Figure (c) must exist in any legal state of the SUPPLY relation that also contains the tuples above the dashed line. Figure (d) shows how the SUPPLY relation with the join dependency is decomposed into three relations R1, R2, and R3 that are each in 5NF. Notice that applying a natural join to any two of these relations produces spurious tuples, but applying a natural join to all three together does not. This is because only the JD exists, but no MVDs are specified. Notice, too, that the JD(R1, R2, R3) is specified on all legal relation states, not just on the one shown in Figure (c). Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer. Therefore, the current practice of database design pays scant attention to them.

INTERNALS OF RDBMS

Physical Data Structures

Databases are stored physically as files of records, which are typically stored on magnetic disks. To organize databases in storage and for accessing them efficiently using various algorithms, require auxiliary data structures called indexes. These structures are often referred to as physical database file structures, and are at the physical level of the three schema architecture.

The collection of data that makes up a computerized database must be stored physically on some computer storage medium. The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a storage hierarchy that includes two main categories:

■ **Primary storage**. This category includes storage media that can be operated on directly by the computer's central processing unit (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity. Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than secondary and tertiary storage devices.

■ **Secondary and tertiary storage**. This category includes magnetic disks, optical disks (CD-ROMs, DVDs, and other similar storage media), and tapes. Hard-disk drives are classified as secondary storage, whereas removable media such as optical disks and tapes are considered tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

Memory Hierarchies and Storage Devices

At the primary storage level, the memory hierarchy includes at the most expensive end, cache memory, which is a static RAM (Random Access Memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining. The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called main memory. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility and lower speed compared with static RAM. At the secondary and tertiary storage level, the hierarchy includes magnetic disks, as well as mass storage in the form of CD-ROM (Compact Disk–Read-Only Memory) and DVD (Digital Video Disk or Digital Versatile Disk) devices, and finally tapes at the least expensive end of the hierarchy. The storage capacity is measured in kilobytes (Kbyte or 1000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1000 GB). The word petabyte (1000 terabytes or 10**15 bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, (magnetic disks), and portions of the database are read into and written from buffers in main memory as needed. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), leading to main memory databases; these are particularly useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.

Between DRAM and magnetic disk storage, another form of memory, flash memory, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memory cards are appearing as the data storage medium in appliances with capacities ranging from a few megabytes to a few gigabytes. These are appearing in cameras, MP3 players, cell phones, PDAs, and so on. USB (Universal

Serial Bus) flash drives have become the most portable medium for carrying data between personal computers; they have a flash memory storage device integrated with a USB interface.

Storage of Databases

Databases typically store large amounts of data that must persist over long periods of time, and hence is often referred to as persistent data. Parts of this data are accessed and processed repeatedly during this period. This contrasts with the notion of transient data that persist for only a limited time during program execution. Most databases are stored permanently (or persistently) on magnetic disk secondary storage, for the following reasons:

■ Generally, databases are too large to fit entirely in main memory.

■ The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage.

■ The cost of storage per unit of data is an order of magnitude less for disk secondary storage than for primary storage. Some of the newer technologies—such as optical disks, DVDs, and tape jukeboxes— are likely to provide viable alternatives to the use of magnetic disks.

The techniques used to store large amounts of structured data on disk are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific DBMS. The process of physical database design involves choosing the particular data organization techniques that best suit the given application requirements from among the options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as files of records. Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently when they are needed. There are several primary file organizations, which determine how the file records are physically placed on the disk, and hence how the records can be accessed. A heap file (or unordered file) places the records on disk in no particular order by appending new records at the end of the file, whereas a sorted file (or sequential file) keeps the records ordered by the value of a particular field (called the sort key). A hashed file uses a hash function applied to a particular field (called the hash key) to determine a record’s placement on disk. Other primary file organizations, such as B-trees, use tree structures.

A secondary organization or auxiliary access structure allows efficient access to file records based on alternate fields than those that have been used for the primary file organization. Most of these exist as indexes.

Files, Fixed-Length Records, and Variable-Length Records

A file is a sequence of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records. If different records in the file have different sizes, the file is said to be made up of variable-length records.

For variable-length fields, each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special separator characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields, or we can store the length in bytes of the field in the record, preceding the field value.

A file of records with optional fields can be formatted in different ways. If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values.

A repeating field needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field.

Finally, for a file that includes records of different types, each record is preceded by a record type indicator.

Query Optimization

A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated. The scanner identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query, whereas the parser checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be validated by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created, usually as a tree data structure called a query tree. It is also possible to represent the query using a graph data structure called a query graph. The DBMS must then devise an execution strategy or query plan for retrieving the results of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as query optimization.

Figure shows the different steps of processing a high-level query. The query optimizer module has the task of producing a good execution plan, and the code generator generates the code to execute that plan. The runtime database processor has the task of running (executing) the query code, whether in compiled or interpreted mode, to

produce the query result. If a runtime error results, an error message is generated by the runtime database processor.

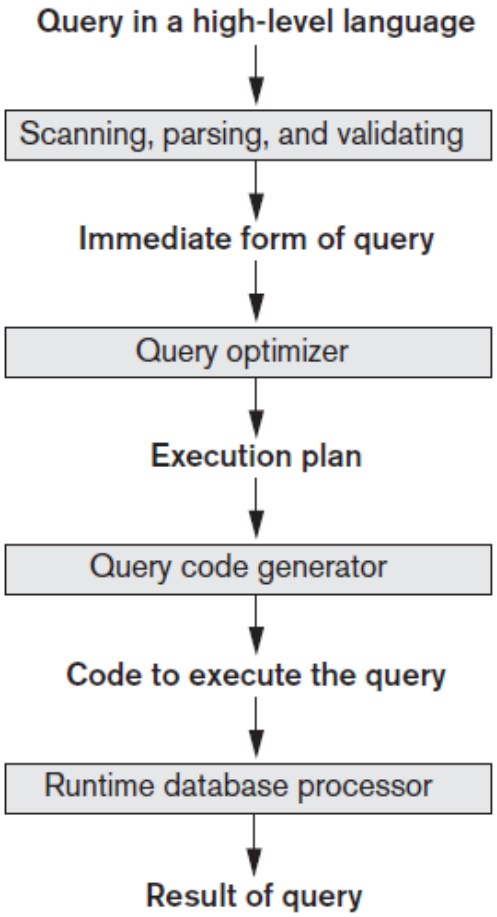


Figure
Typical steps when processing a high-level query.

Translating SQL Queries into Relational Algebra

In practice, SQL is the query language that is used in most commercial RDBMSs.An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra.

Consider the following SQL query on the EMPLOYEE relation:

```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ( SELECT MAX (Salary)
```

```
FROM EMPLOYEE
WHERE Dno=5 );
```

The inner block could be translated into the following extended relational algebra expression:

$$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

and the outer block into the expression:

$$\pi_{\text{Lname,Fname}}(\sigma_{\text{Salary>c}}(\text{EMPLOYEE}))$$

The query optimizer would then choose an execution plan for each query block.

Algorithms for External Sorting

Sorting is one of the primary algorithms used in query processing. For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted. Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (such as UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause). Note that sorting of a particular file may be avoided if an appropriate index—such as a primary or clustering index—exists on the desired file attribute to allow ordered access to the records of the file. External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory.

The typical external sorting algorithm uses a sort-merge strategy, which starts by sorting small subfiles—called runs—of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.

Algorithms for SELECT and JOIN Operations

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions. We will use the following operations, specified on the relational database for illustration:

OP1: $\sigma_{\text{Ssn = '123456789'}}$ (EMPLOYEE)

OP2: $\sigma_{\text{Dnumber > 5}}$ (DEPARTMENT)

OP3: $\sigma_{\text{Dno = 5}}$ (EMPLOYEE)

OP4: $\sigma_{\text{Dno = 5 AND Salary > 30000 AND Sex = 'F'}}$ (EMPLOYEE)

OP5: $\sigma_{\text{Essn='123456789' AND Pno =10}}$(WORKS_ON)

Search Methods for Simple Selection

A number of search algorithms are possible for selecting records from a file. These are also known as *file scans*, because they scan the records of a file to search for and retrieve records that satisfy a selection condition. If the search algorithm involves the use of an index, the index search is called an *index scan*. The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

■ **S1—Linear search (brute force algorithm)**. Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition. Since the records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

■ **S2—Binary search**. If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search—which is more efficient than linear search—can be used. An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.

■ **S3a—Using a primary index**. If the selection condition involves an equality comparison on a key attribute with a primary index—for example, Ssn = '123456789' in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).

■ **S3b—Using a hash key**. If the selection condition involves an equality comparison on a key attribute with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).

■ **S4—Using a primary index to retrieve multiple records**. If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5), then retrieve all subsequent records in the (ordered) file. For the condition Dnumber < 5, retrieve all the preceding records.

■ **S5—Using a clustering index to retrieve multiple records**. If the selection condition involves an equality comparison on a nonkey attribute with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.

■ **S6—Using a secondary (B+-tree) index on an equality comparison**. This search method can be used to retrieve a single record if the indexing field is a key (has unique values) or to retrieve multiple records if the indexing field is not a key. This can also be used for comparisons involving >, >=, <, or <=.

Search Methods for Complex Selection

If a condition of a SELECT operation is a conjunctive condition—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

■ **S7—Conjunctive selection using an individual index**. If an attribute involved in any single simple condition in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive select condition.

■ **S8—Conjunctive selection using a composite index**. If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields— for example, if an index has been created on the composite key (Essn, Pno) of the WORKS_ON file for OP5—we can use the index directly.

■ **S9—Conjunctive selection by intersection of record pointers**. If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the set of record pointers that satisfy the individual condition. The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions. In general, method S9 assumes that each of the indexes is on a nonkey field of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition.

Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever more than one of the attributes involved in the conditions have an access path. The optimizer should choose the access path that retrieves the fewest records in the most efficient way by estimating the different costs and choosing the method with the least estimated cost.

Implementing the JOIN Operation

The JOIN operation is one of the most time-consuming operations in query processing. There are many possible ways to implement a two-way join, which is a join on two files. The algorithms we discuss next are for a join operation of the form:

$$R \bowtie_{A=B} S$$

where A and B are the join attributes, which should be domain-compatible attributes of R and S, respectively. We illustrate four of the most common techniques for performing such a join, using the following sample operations:

OP6: EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT

OP7: DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE

Methods for Implementing Joins

■ **J1—Nested-loop join (or nested-block join)**. This is the default (brute force) algorithm, as it does not require any special access paths on either file in the join. For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].

■ **J2—Single-loop join (using an access structure to retrieve the matching records)**. If an index (or hash key) exists for one of the two join attributes—say, attribute B of file S—retrieve each record t in R (loop over file R), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records s from S that satisfy s[B] = t[A].

■ **J3—Sort-merge join**. If the records of R and S are physically sorted (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they may be sorted first by using external sorting.

■ **J4—Partition-hash join**. The records of files R and S are partitioned into smaller files. The partitioning of each file is done using the same hashing function h on the join attribute A of R (for partitioning file R) and B of S (for partitioning file S). First, a single pass through the file with fewer records (say,R) hashes its records to the various partitions of R; this is called the partitioning phase, since the records of R are partitioned into the hash buckets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of R are all kept in main memory. The collection of records with the same value of h(A) are placed in the same partition, which is a hash bucket in a hash table in main memory. In the second phase, called the probing phase, a single pass through the other file (S) then hashes each of its records using the same hash function h(B) to probe the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files fits entirely into memory buckets after the first phase. In practice, techniques J1 to J4 are implemented by accessing whole disk blocks of a file, rather than individual records. Depending on the available number of buffers in memory, the number of blocks read in from the file can be adjusted.

Implementing OUTER JOINs

Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join. For example, to compute a left outer join, we use the left relation as the outer loop or single-loop because every tuple in the left relation must appear in the result. If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with NULL value(s). The sort-merge and hash-join algorithms can also be extended to compute outer joins.

Theoretically, outer join can also be computed by executing a combination of relational algebra operators. For example, the left outer join operation shown above is equivalent to the following sequence of relational operations:

1. Compute the (inner) JOIN of the EMPLOYEE and DEPARTMENT tables.

$$TEMP1 \leftarrow \pi_{Lname,\ Fname,\ Dname}\ (EMPLOYEE\ \bowtie_{Dno=Dnumber}\ DEPARTMENT)$$

2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.

$$TEMP2 \leftarrow \pi_{Lname,\ Fname}\ (EMPLOYEE) - \pi_{Lname,\ Fname}\ (TEMP1)$$

3. Pad each tuple in TEMP2 with a NULL Dname field.

$$TEMP2 \leftarrow TEMP2 \times NULL$$

4. Apply the UNION operation to TEMP1, TEMP2 to produce the LEFT OUTER JOIN result.

$$RESULT \leftarrow TEMP1 \cup TEMP2$$

Combining Operations Using Pipelining

A query specified in SQL will typically be translated into a relational algebra expression that is a sequence of relational operations. If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead. Generating and storing large temporary files on disk is time-consuming and can be unnecessary in many cases, since these files will immediately be used as input to the next operation. To reduce the number of temporary files, it is common to generate query execution code that corresponds to algorithms for combinations of operations in a query.

For example, rather than being implemented separately, a JOIN can be combined with two SELECT operations on the input files and a final PROJECT operation on the resulting file; all this is implemented by one algorithm with two input files and a single output file. Rather than creating four temporary files, we apply the algorithm directly and get just one result file. This is called pipelining or stream-based processing.

It is common to create the query execution code dynamically to implement multiple operations. The generated code for producing the query combines several algorithms that correspond to individual operations. As the result tuples from one operation are produced, they are provided as input for subsequent operations. For example, if a join operation follows two select operations on base relations, the tuples resulting from each select are provided as input for the join algorithm in a stream or pipeline as they are produced.

Using Heuristics in Query Optimization

The scanner and parser of an SQL query first generate a data structure that corresponds to an initial query representation, which is then optimized according to heuristic rules. This leads to an optimized query representation, which corresponds to the query execution strategy. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied before a join or other binary operation.

A query tree is used to represent a relational algebra or extended relational algebra expression, whereas a query graph is used to represent a relational calculus expression.

Notation for Query Trees and Query Graphs

A query tree is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations starts at the leaf nodes, which represents the input database relations for the query, and ends at the root node, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.

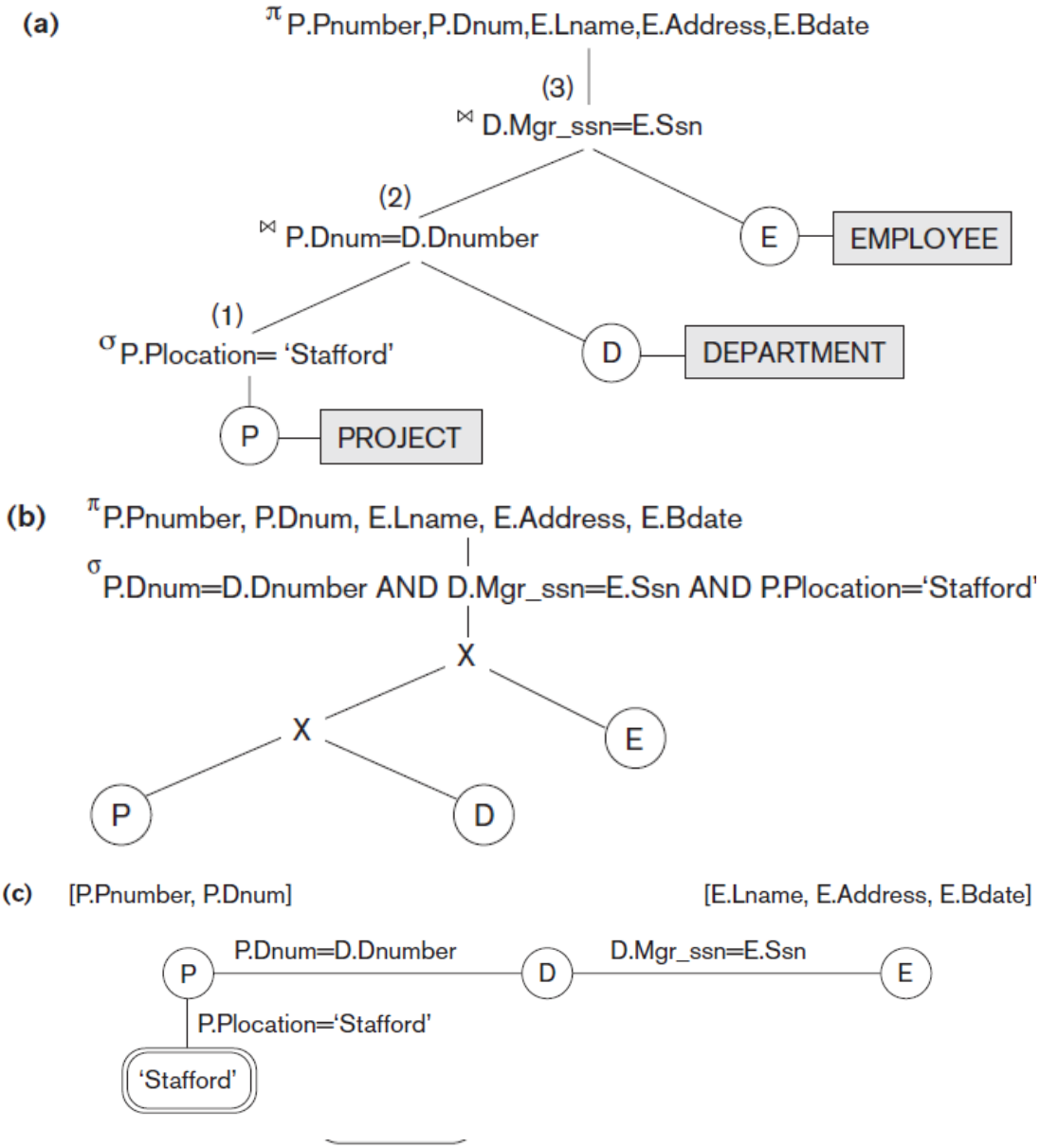
Figure a shows a query tree for query: *For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate.* This query is specified on the COMPANY relational schema and corresponds to the following relational algebra expression:

$$\pi_{Pnumber,\ Dnum,\ Lname,\ Address,\ Bdate}\ (((\sigma_{Plocation='Stafford'}(PROJECT))$$

$$\bowtie_{Dnum=Dnumber}(DEPARTMENT)) \ \bowtie_{Mgr\_ssn=Ssn}(EMPLOYEE))$$

This corresponds to the following SQL query:

```
SELECT   P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM     PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE    P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
         P.Plocation= 'Stafford';
```



Figure

Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

In Figure a, the leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively, and the internal tree nodes represent the relational algebra operations of the expression. When this query tree is executed, the node marked (1) in Figure a must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the query graph notation. Figure c shows the query graph for query Q2. Relations in the query are represented by relation nodes, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by constant nodes, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph edges, as shown in Figure c. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query. Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

Heuristic Optimization of Query Trees

In general, many different relational algebra expressions—and hence many different query trees—can be equivalent; that is, they can represent the same query.

The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization. For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure (b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is very inefficient if executed directly, because of the CARTESIAN PRODUCT (×) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. The heuristic query optimizer will transform this initial query tree into an equivalent final query tree that is efficient to execute.

The optimizer must include rules for equivalence among relational algebra expressions that can be applied to transform the initial tree into the final, optimized query tree.

Example of Transforming a Query.

Consider the following query Q: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'*. This query can be specified in SQL as follows:

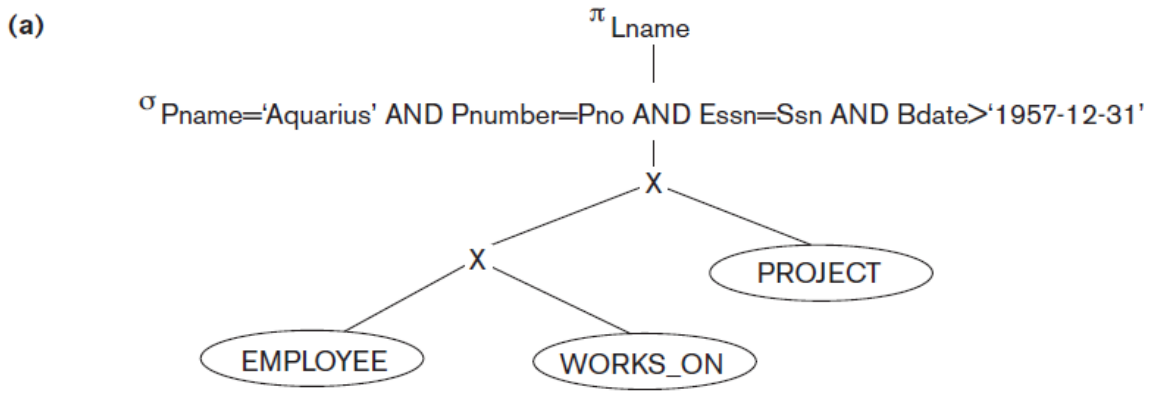
```
Q:  SELECT   Lname
    FROM     EMPLOYEE, WORKS_ON, PROJECT
    WHERE    Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn
             AND Bdate > '1957-12-31';
```

The initial query tree for Q is shown in Figure (a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to execute. This particular query needs only one record from the PROJECT relation—for the 'Aquarius' project—and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'. Figure (b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

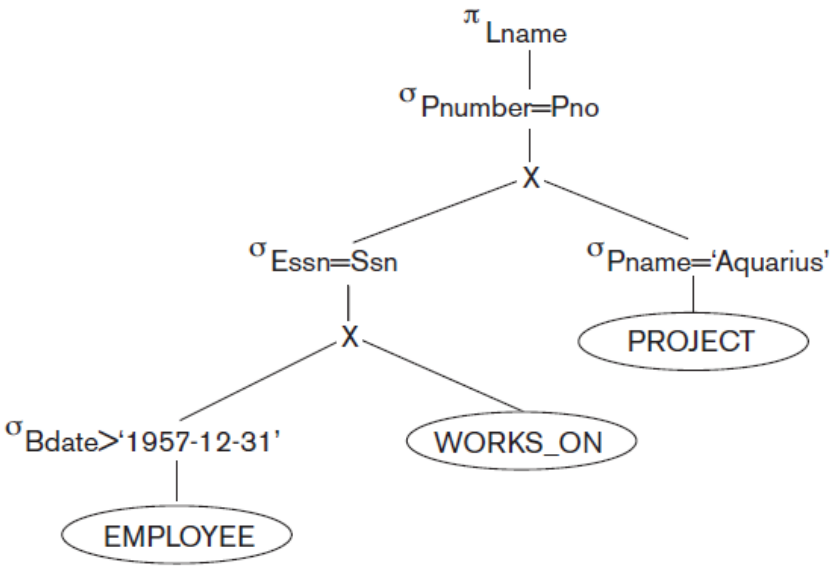
Figure:

Steps in converting a query tree during heuristic optimization.

(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

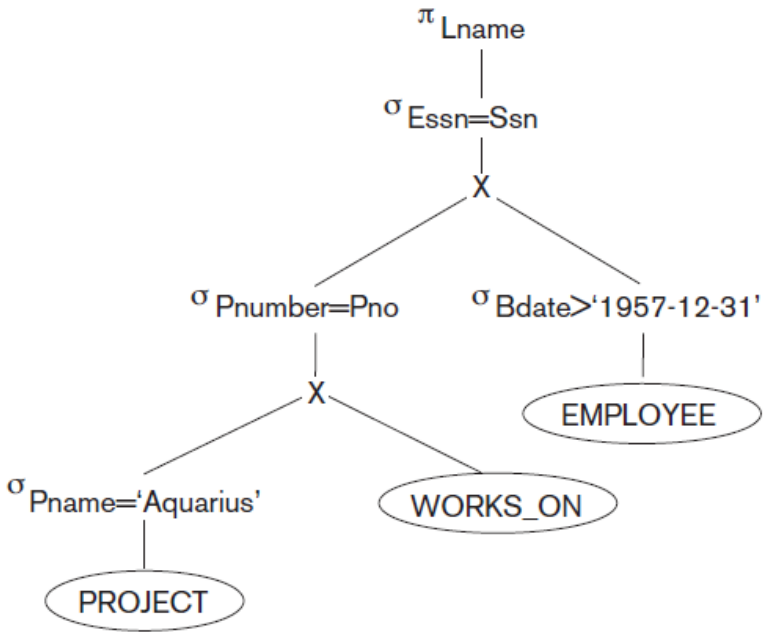


(b)



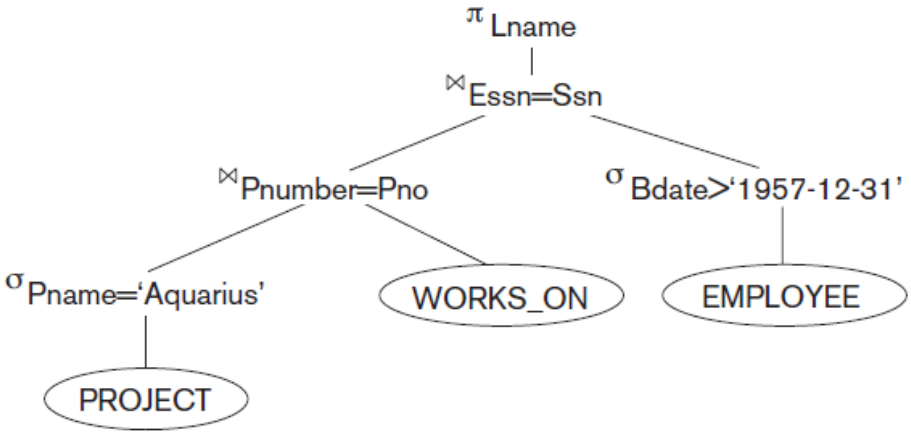
A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure (c). This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only.

(c)



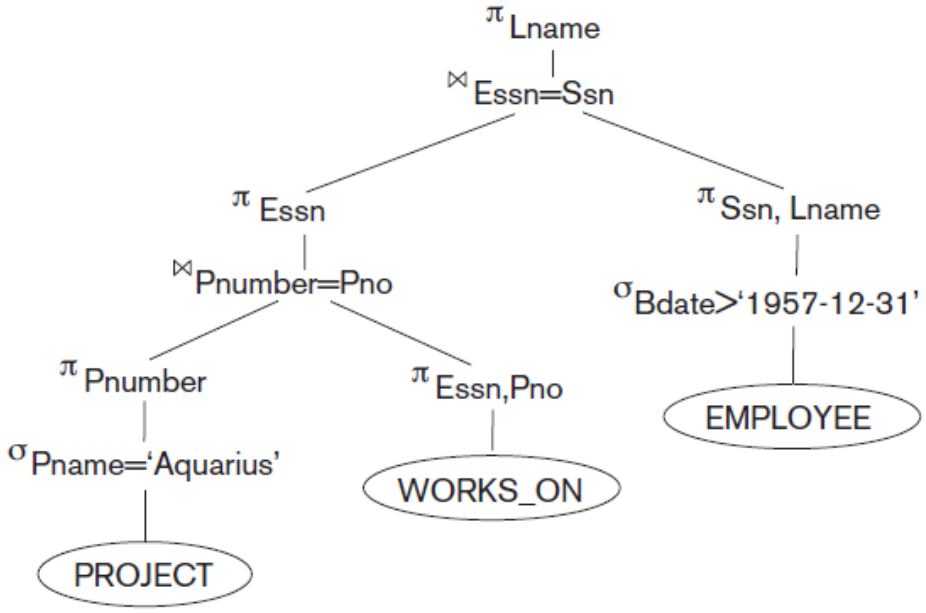
We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure (d).

(d)



Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree, as shown in Figure (e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

(e)



As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules preserve this equivalence.

General Transformation Rules for Relational Algebra Operations

There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of

attributes in a different order but the two relations represent the same information, we consider the relations to be equivalent. We will state some transformation rules that are useful in query optimization, without proving them:

1. **Cascade of σ** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:
$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$$
2. **Commutativity of σ.** The σ operation is commutative:
$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$
3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:
$$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$$
4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:
$$\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$$

5. Commutativity of ⋈ (and ×). The join operation is commutative, as is the × operation:

$R \bowtie_c S \equiv S \bowtie_c R$

$R \times S \equiv S \times R$

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the meaning is the same because the order of attributes is not important in the alternative definition of relation.

6. Commuting σ with ⋈ (or ×). If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

$\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$

Alternatively, if the selection condition c can be written as (c1 AND c2), where condition c1 involves only the attributes of R and condition c2 involves only the attributes of S, the operations commute as follows:

$\sigma_c (R \bowtie S) \equiv (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$

The same rules apply if the is replaced by a × operation.

Outline of a Heuristic Algebraic Optimization Algorithm

We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure. The steps of the algorithm are as follows:

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.

2. Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from only one table, which means that it represents a selection condition, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from two tables, which means that it represents a join condition, the condition is moved to a location down the tree after the two tables are combined.

3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of most restrictive SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size. Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.

4. Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.

5. Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure (b) shows the tree in Figure (a) after applying steps 1 and 2 of the algorithm; Figure (c) shows the tree after step 3; Figure (d) after step 4; and Figure (e) after step 5. In step 6 we may group together the operations in the subtree whose root is the operation π_{Essn} into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation π_{Essn}, because the first grouping means that this subtree is executed first.

Summary of Heuristics for Algebraic Optimization

The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

Converting Query Trees into Query Execution Plans

An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1, whose corresponding relational algebra expression is

$$\pi_{Fname,\ Lname,\ Address}(\sigma_{Dname='Research'}(DEPARTMENT) \bowtie_{Dnumber=Dno} EMPLOYEE)$$

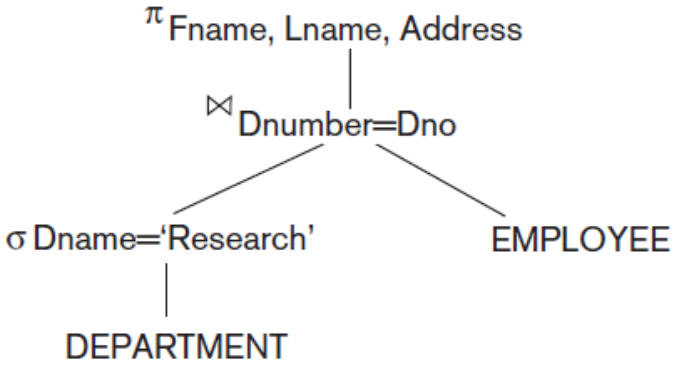


Fig. A Query Tree for Q1

The query tree is shown in Figure. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), a single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

With materialized evaluation, the result of an operation is stored as a temporary relation (that is, the result is physically materialized). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand, with pipelined evaluation, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

Using Selectivity and Cost Estimates in Query Optimization

A query optimizer does not depend solely on heuristic rules; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the lowest cost estimate. For this approach to work, accurate cost estimates are required so that different strategies can be compared fairly and realistically. In addition, the optimizer must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for compiled queries where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For interpreted queries, where the entire process occurs at runtime, a full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

This approach is generally referred to as cost-based query optimization. It uses traditional optimization techniques that search the solution space to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one.

Cost Components for Query Execution

1. Access cost to secondary storage. This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as disk I/O (input/output) cost. The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

2. Disk storage cost. This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.

3. Computation cost. This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as CPU (central processing unit) cost.

4. Memory usage cost. This is the cost pertaining to the number of main memory buffers needed during query execution.

5. Communication cost. This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases, it would also include the cost of transferring tables and results among various computers during query evaluation.

For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory buffers. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved, communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access.

Catalog Information Used in Cost Functions

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the number of records (tuples) (r), the (average) record size (R), and the number of file blocks (b) are

needed. The blocking factor (bfr) for the file may also be needed.We must also keep track of the primary file organization for each file.

The primary file organization records may be unordered, ordered by an attribute with or without a primary or clustering index, or hashed. Information is also kept on all primary, secondary, or clustering indexes and their indexing attributes.

Another important parameter is the number of distinct values (d) of an attribute and the attribute selectivity (sl), which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the selection cardinality (s = sl*r) of an attribute, which is the average number of records that will satisfy an equality selection condition on that attribute. For a key attribute, d = r, sl = 1/r and s = 1. For a nonkey attribute, by making an assumption that the d distinct values are uniformly distributed among the records, we estimate sl = (1/d) and so s = (r/d). The query optimizer will need reasonably close but not necessarily completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies.

Examples of Cost Functions for SELECT

We now give cost functions for the selection algorithms S1 to S8 discussed Previously in terms of number of block transfers between memory and disk. These cost functions are estimates that ignore computation time, storage cost, and other factors. The cost for method Si is referred to as C_{Si} block accesses.

■ **S1—Linear search (brute force) approach**. We search all the file blocks to retrieve all records satisfying the selection condition; hence, $C_{S1a} = b$. For an equality condition on a key attribute, only half the file blocks are searched on the average before finding the record, so a rough estimate for $C_{S1b} = (b/2)$ if the record is found; if no record is found that satisfies the condition, $C_{S1b} = b$.

■ **S2—Binary search**. This search accesses approximately $C_{S2} = \log_2 b + (s/bfr)-1$ file blocks. This reduces to $\log_2 b$ if the equality condition is on a unique (key) attribute, because s = 1 in this case.

■ **S3a—Using a primary index to retrieve a single record**. For a primary index, retrieve one disk block at each index level, plus one disk block from the data file. Hence, the cost is one more disk block than the number of index levels: $CS_{3a} = x + 1$.

■ **S3b—Using a hash key to retrieve a single record**. For hashing, only one disk block needs to be accessed in most cases. The cost function is approximately $C_{S3b} = 1$ for static hashing or linear hashing.

■ **S4—Using an ordering index to retrieve multiple records**. If the comparison condition is >, >=, <, or <= on a key field with an ordering index, roughly half the file

records will satisfy the condition. This gives a cost function of CS4 = x + (b/2). This is a very rough estimate, and although it may be correct on the average, it may be quite inaccurate in individual cases. A more accurate estimate is possible if the distribution of records is stored in a histogram.

■ **S5—Using a clustering index to retrieve multiple records**. One disk block is accessed at each index level, which gives the address of the first file disk block in the cluster. Given an equality condition on the indexing attribute, s records will satisfy the condition, where s is the selection cardinality of the indexing attribute. This means that (s/bfr) file blocks will be in the cluster of file blocks that hold all the selected records, giving C_{S5} = x + (s/bfr).

■ **S6—Using a secondary (B+-tree) index**. For a secondary index on a key (unique) attribute, the cost is x + 1 disk block accesses. For a secondary index on a nonkey (nonunique) attribute, s records will satisfy an equality condition, where s is the selection cardinality of the indexing attribute. However, because the index is nonclustering, each of the records may reside on a different disk block, so the (worst case) cost estimate is C_{S6a} = x + 1 + s. The additional 1 is to account for the disk block that contains the record pointers after the index is searched. If the comparison condition is >, >=, <, or <= and half the file records are assumed to satisfy the condition, then (very roughly) half the first-level index blocks are accessed, plus half the file records via the index. The cost estimate for this case, approximately, is C_{S6b} = x + (b_{I1}/2) + (r/2). The r/2 factor can be refined if better selectivity estimates are available through a histogram. The latter method C_{S6b} can be very costly.

■ **S7—Conjunctive selection**. We can use either S1 or one of the methods S2 to S6 discussed above. In the latter case, we use one condition to retrieve the records and then check in the main memory buffers whether each retrieved record satisfies the remaining conditions in the conjunction. If multiple indexes exist, the search of each index can produce a set of record pointers (record ids) in the main memory buffers. The intersection of the sets of record pointers (referred to in S9) can be computed in main memory, and then the resulting records are retrieved based on their record ids.

■ S8—Conjunctive selection using a composite index. Same as S3a, S5, or S6a, depending on the type of index.

Example of Using the Cost Functions

Suppose that the EMPLOYEE file has r_E = 10,000 records stored in b_E = 2000 disk blocks with blocking factor bfr_E = 5 records/block and the following access paths:

1. A clustering index on Salary, with levels x_{Salary} = 3 and average selection cardinality s_{Salary} = 20. (This corresponds to a selectivity of sl_{Salary} = 0.002).

2. A secondary index on the key attribute Ssn, with $x_{Ssn} = 4$ ($s_{Ssn} = 1$, $sl_{Ssn} = 0.0001$).

3. A secondary index on the nonkey attribute Dno, with $x_{Dno} = 2$ and first-level index blocks $b_{I1Dno} = 4$. There are $d_{Dno} = 125$ distinct values for Dno, so the selectivity of Dno is $sl_{Dno} = (1/d_{Dno}) = 0.008$, and the selection cardinality is $s_{Dno} = (r_E * sl_{Dno}) = (r_E/d_{Dno}) = 80$.

4. A secondary index on Sex, with $x_{Sex} = 1$. There are $d_{Sex} = 2$ values for the Sex attribute, so the average selection cardinality is $s_{Sex} = (r_E/d_{Sex}) = 5000$.

We illustrate the use of cost functions with the following examples:

OP1: $\sigma_{Ssn='123456789'}(EMPLOYEE)$

OP2: $\sigma_{Dno>5}(EMPLOYEE)$

OP3: $\sigma_{Dno=5}(EMPLOYEE)$

OP4: $\sigma_{Dno=5\ AND\ SALARY>30000\ AND\ Sex='F'}(EMPLOYEE)$

The cost of the brute force (linear search or file scan) option S1 will be estimated as$C_{S1a} = b_E = 2000$ (for a selection on a nonkey attribute) or $C_{S1b} = (b_E/2) = 1000$ (average cost for a selection on a key attribute). For OP1 we can use either method S1 or method S6a; the cost estimate for S6a is $C_{S6a} = x_{Ssn} + 1 = 4 + 1 = 5$, and it is chosen over method S1, whose average cost is $C_{S1b} = 1000$. For OP2 we can use either method S1 (with estimated cost $C_{S1a} = 2000$) or method S6b (with estimated cost $C_{S6b} = x_{Dno} + (b_{I1Dno}/2) + (r_E/2) = 2 + (4/2) + (10{,}000/2) = 5004$), so we choose the linear search approach for OP2. For OP3 we can use either method S1 (with estimated cost $C_{S1a} = 2000$) or method S6a (with estimated cost $C_{S6a} = x_{Dno} + s_{Dno} = 2 + 80 = 82$), so we choose method S6a.

Finally, consider OP4, which has a conjunctive selection condition. We need to estimate the cost of using any one of the three components of the selection condition to retrieve the records, plus the linear search approach. The latter gives cost estimate $C_{S1a} = 2000$. Using the condition (Dno = 5) first gives the cost estimate $C_{S6a} = 82$. Using the condition (Salary > 30,000) first gives a cost estimate $C_{S4} = x_{Salary} + (b_E/2) = 3 + (2000/2) = 1003$.Using the condition (Sex = 'F') first gives a cost estimate $C_{S6a} = x_{Sex} + s_{Sex} = 1 + 5000 = 5001$. The optimizer would then choose method S6a on the secondary index on Dno because it has the lowest cost estimate. The condition (Dno = 5) is used to retrieve the records, and the remaining part of the conjunctive condition (Salary > 30,000 AND Sex = 'F') is checked for each selected record after it is retrieved into memory. Only the records that satisfy these additional conditions are included in the result of the operation.

UNIT 4

TRANSACTION PROCESSING, RECOVERY SYSTEM, & CONCURRENCY CONTROL

The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

TRANSACTION PROCESSING

Single-User versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system concurrently. A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system—and hence access the database—concurrently.

A single central processing unit (CPU) can only execute at most one process at a time. However, multiprogramming operating systems execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually interleaved. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes. If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible

Transactions

A transaction is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction; otherwise it is known as a read-write transaction.

Database Items

A database is basically represented as a collection of named data items. The size of a data item is called its granularity. A data item can be a database record, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database. Each data item has a unique name, but this name is not typically used by the programmer; rather, it is just a means to uniquely identify each data item. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name.

Read and Write Operations

Using this simplified database model, the basic database access operations that a transaction can include are as follows:

■ **read_item(X)**. Reads a database item named X into a program variable.

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

Executing a read_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Executing a write_item(X) command includes the following steps:

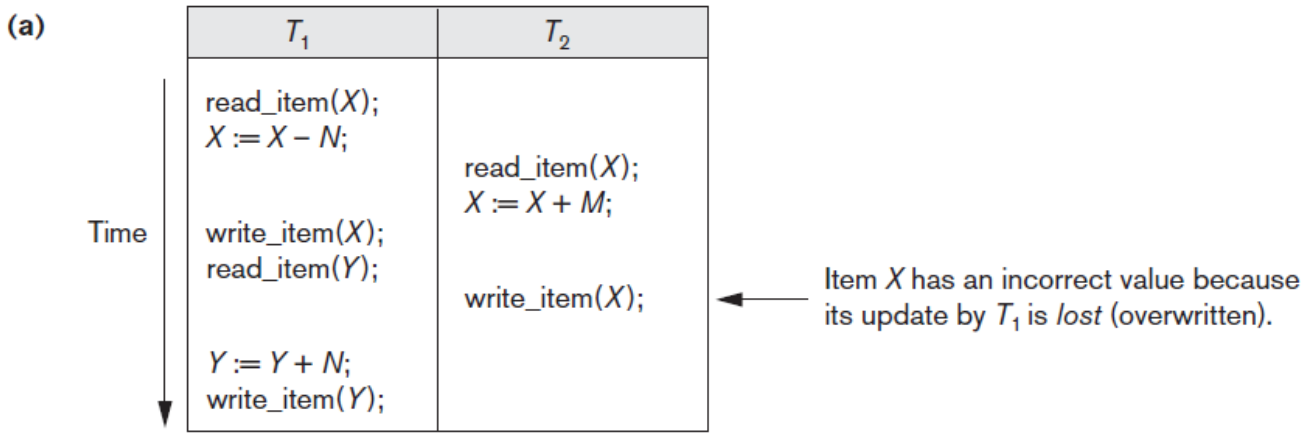
1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.

4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Concurrency Control

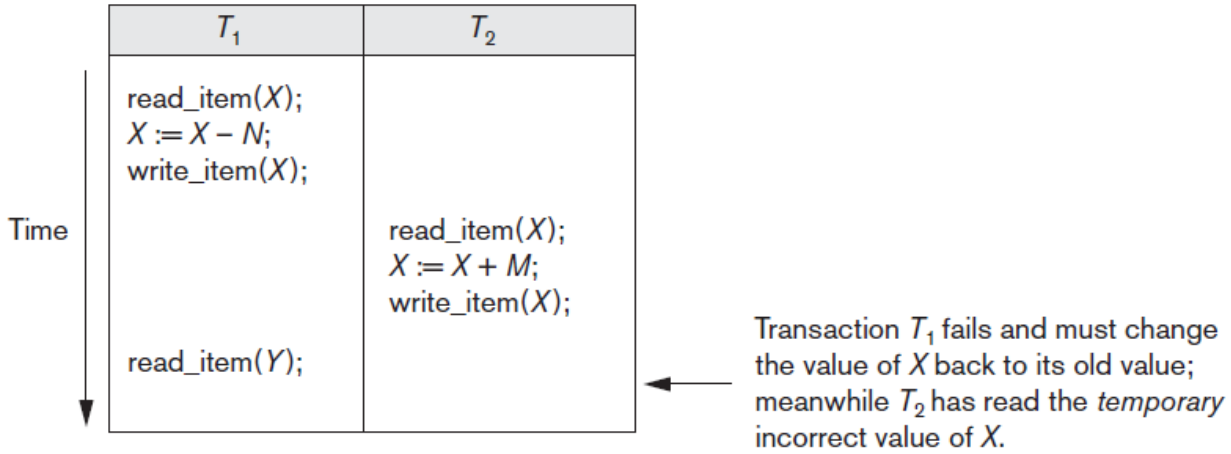
Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems.

The Lost Update Problem: This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure (a); then the final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.



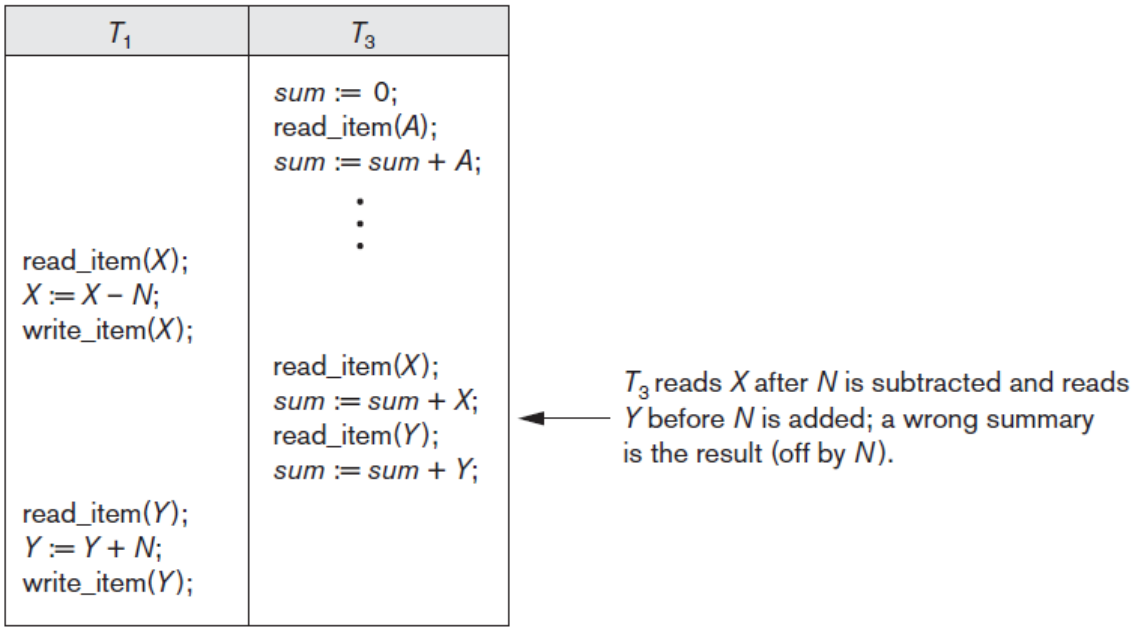
The Temporary Update (or Dirty Read) Problem: This problem occurs when one transaction updates a database item and then the transaction fails for some reason Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure (b) shows an example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1. The value of item X that is read by T2 is called dirty data because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

(b)



The Incorrect Summary Problem: If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure (c) occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

(c)



The Unrepeatable Read Problem: Another problem that may occur is called unrepeatable read, where a transaction T reads the same item twice and the item is changed by another transaction T' between the two reads. Hence, T receives different values for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several

flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

System Recovery

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be committed, whereas in the second case, the transaction is aborted. The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not, because the whole transaction is a logical unit of database processing. If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. A computer failure (system crash): A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

2. A transaction or system error: Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction: During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

4. Concurrency control enforcement: The concurrency control method may decide to abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

5. Disk failure: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes: This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Transaction States and Additional Operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts. Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

■ **BEGIN_TRANSACTION**. This marks the beginning of transaction execution.

■ **READ or WRITE**. These specify read or write operations on the database items that are executed as part of a transaction.

■ **END_TRANSACTION**. This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

■ **COMMIT_TRANSACTION.** This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

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

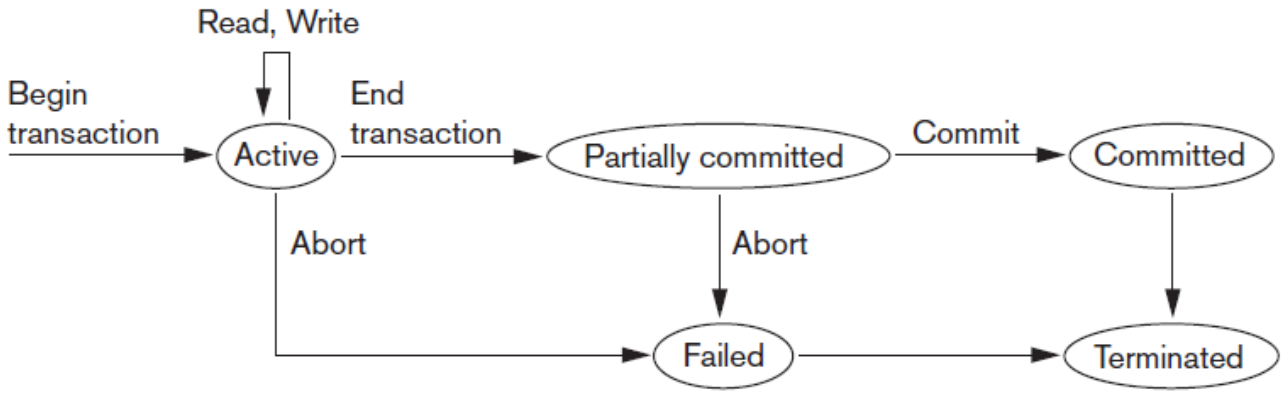


Fig. State Transition Diagram

Figure shows a state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an active state immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the partially committed state. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording change in the system log. Once this check is successful, the transaction is said to have reached its commit point and enters the committed state. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later—either automatically or after being resubmitted by the user—as brand new transactions.

System Log

To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer. When the log buffer is filled, or when certain other conditions occur, the log buffer is appended to the end of the log file on disk. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures. The following are the types of entries—called log records—that are written to the log file and the corresponding action for each log record. In these entries, T refers to a unique transaction-id that is generated automatically by the system for each transaction and that is used to identify each transaction:

1. [start_transaction, T]: Indicates that transaction T has started execution.

2. [write_item, T, X, old_value, new_value] Indicates that transaction T has changed the value of database item X from old_value to new_value.

3. [read_item, T, X] Indicates that transaction T has read the value of database item X.

4. [commit, T] Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

5. [abort, T] Indicates that transaction T has been aborted.

We are assuming that all permanent changes to the database occur within transactions, so the notion of recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log. If the system crashes, we can recover to a consistent database state by examining the log. 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. Redo of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can be sure that all these new_values have been written to the actual database on disk from the main memory buffers.

Commit Point of a Transaction

A transaction T reaches its commit point when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be committed, and its effect must be permanently recorded in the database. The transaction then writes a commit record [commit, T] into the log. If a system failure occurs, we can search back in the log for all transactions T that have written a [start_transaction, T] record into the log but have not written their [commit, T] record yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be redone from the log records.

The log file must be kept on disk. Updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk. It is common to keep one or more blocks of the log file in main memory buffers, called the log buffer, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file buffer. 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. Hence, 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 buffer before committing a transaction.

Properties of Transactions ACID

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

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

■ **Consistency preservation:** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

■ **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

■ **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the complete execution of the transaction, assuming that no interference with other transactions occurs.

The isolation property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems. There have been attempts to define the level of isolation of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called true isolation) has, in addition to level 2 properties, repeatable reads.

And last, the durability property is the responsibility of the recovery subsystem of the DBMS.

Schedule

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule (or history).

A schedule (or history) S of n transactions T1, T2, ..., Tn is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule S. However, for each transaction Ti that participates in the schedule S, the operations of Ti in S must appear in the same order in which they occur in Ti. The order of operations in S is considered to be a total ordering, meaning that for any two operations in the schedule, one must occur before the other.

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions: (1) they belong to different transactions; (2) they access the same item X; and (3) at least one of the operations is a write_item(X). For example, in schedule Sa, the operations r1(X) and w2(X) conflict, as do the operations r2(X) and w1(X), and the operations w1(X) and w2(X). However, the operations r1(X) and r2(X) do not conflict, since they are both read operations; the operations w2(X) and w1(Y) do not conflict because they operate on distinct data items X and Y; and the operations r1(X) and w1(X) do not conflict because they belong to the same transaction.

Intuitively, two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations r1(X); w2(X) to w2(X); r1(X), then the value of X that is read by transaction T1 changes, because in the second order the value of X is changed by w2(X) before it is read by r1(X), whereas in the first order the value is read before it is changed. This is called a read-write conflict. The other type is called a write-write conflict, and is illustrated by the case where we change the order of two operations such as w1(X); w2(X) to w2(X); w1(X). For a write-write conflict, the last value of X will differ because in one case it is written by T2 and in the other case by T1.

Serial schedule

Schedule that does not interleave the actions of different transactions. In schedule 1 shown in fig. first all the instructions of T1 are grouped and run together. Then all the instructions of T2 are grouped and run together. Means schedule 2 will not start until all the instructions of schedule 1 are complete. This type of schedules is called serial schedule.

Recoverability

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written some item X that T reads have committed. A transaction T reads from transaction T' in a schedule S if some item X is first written by T' and later read by T. In addition, T' should not have been aborted before T reads item X, and there should be no transactions that write X after T' writes it and before T reads it (unless those transactions, if any, have aborted before T reads X).

Serializability

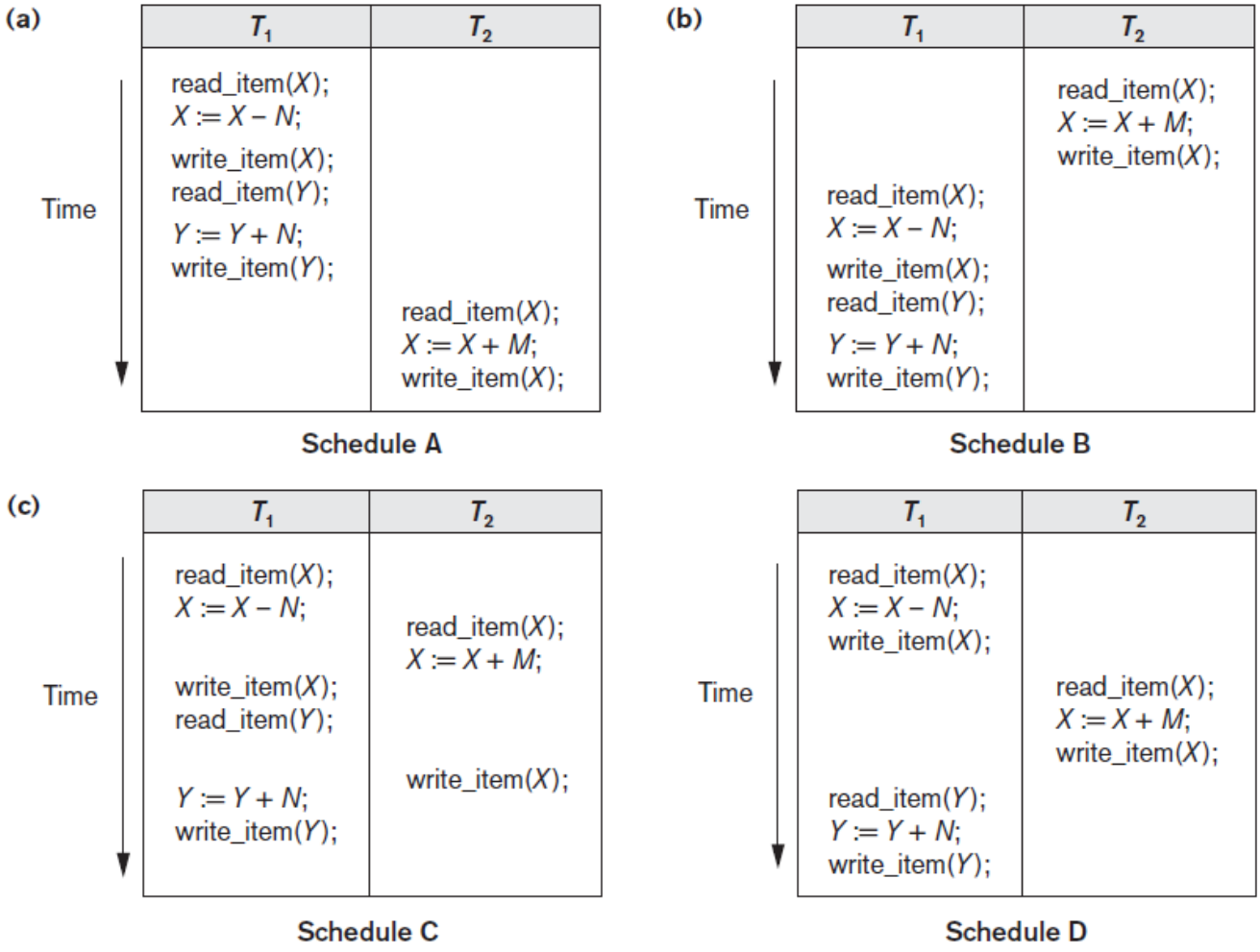
Schedules that are always considered to be correct when concurrent transactions are executing. Such schedules are known as serializable schedules. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T1 and T2 in Figure at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).

2. Execute all the operations of transaction T2 (in sequence) followed by all the

operations of transaction T1 (in sequence).

These two schedules—called serial schedules—are shown in Figure (a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown in Figure (c). The concept of serializability of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.



Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figure (a) and (b) are called serial because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order.

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

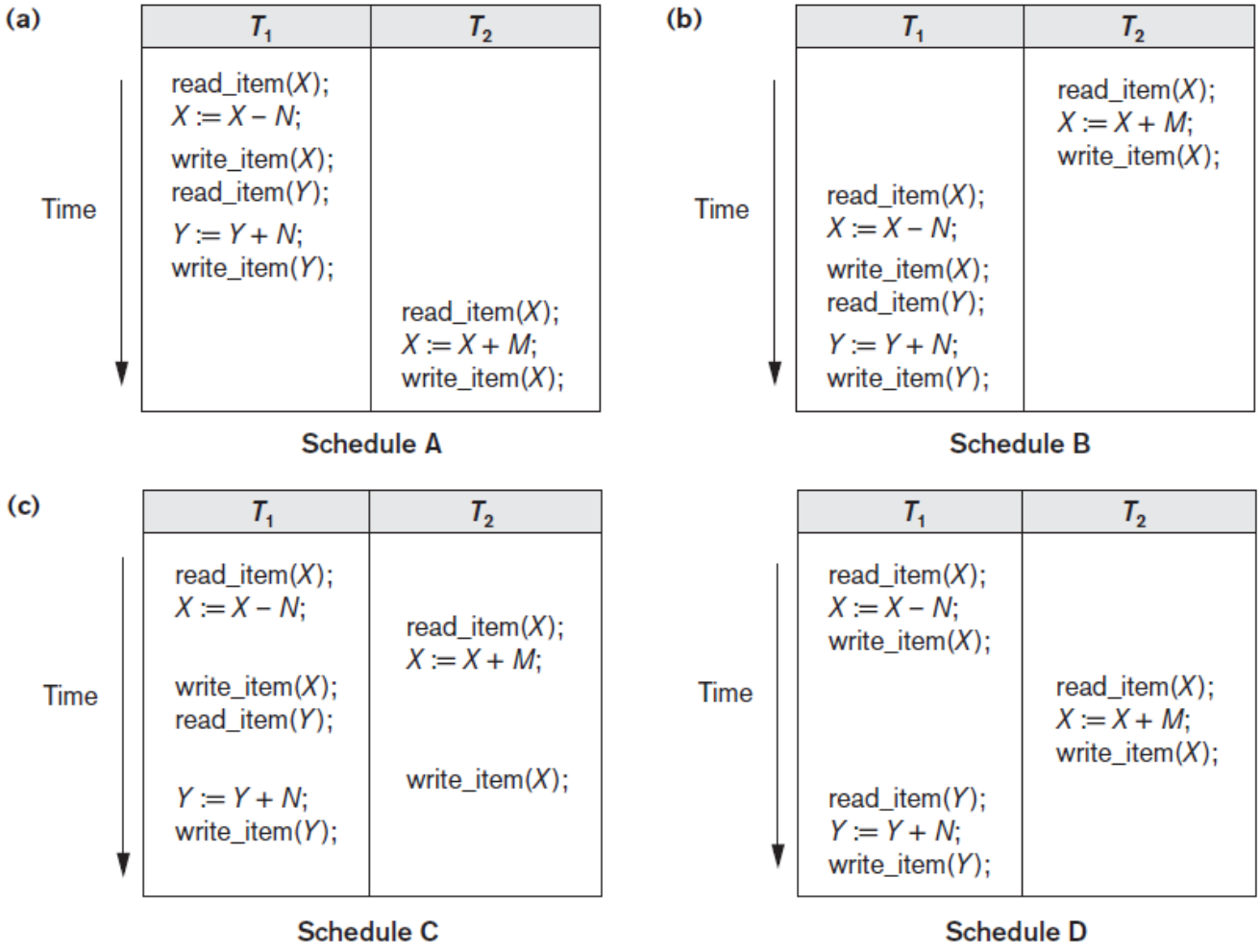
The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Additionally, if some transaction T is quite long, the other transactions must wait for T to complete all its operations before starting. Hence, serial schedules are considered unacceptable in practice. However, if we can determine which other schedules are equivalent to a serial schedule, we can allow these schedules to occur.

A schedule S of n transactions is *serializable* if it is equivalent to some serial schedule of the same n transactions. There are n! possible serial schedules of n transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules—those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to any serial schedule and hence are not serializable.

There are several ways to define schedule equivalence. The simplest but least satisfactory definition involves comparing the effects of the schedules on the database. Two schedules are called result equivalent if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. Hence, result equivalence alone cannot be used to define equivalence of schedules. Two definitions of equivalence of schedules are generally used: *conflict equivalence* and *view equivalence*.

Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules. If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent.

Using the notion of conflict equivalence, we define a schedule S to be conflict serializable if it is (conflict) equivalent to some serial schedule S'. In such a case, we can reorder the non conflicting operations in S until we form the equivalent serial schedule S'. According to this definition, schedule D in Figure (c) is equivalent to the serial schedule A in Figure (a). In both schedules, the read_item(X) of T2 reads the value of X written by T1, while the other read_item operations read the database values from the initial database state. Additionally, T1 is the last transaction to write Y, and T2 is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. Notice that the operations r1(Y) and w1(Y) of schedule D do not conflict with the operations r2(X) and w2(X), since they access different data items. Therefore, we can move r1(Y), w1(Y) before r2(X), w2(X), leading to the equivalent serial schedule T1, T2.



Schedule C in Figure (c) is not equivalent to either of the two possible serial schedules A and B, and hence is not serializable. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because r2(X) and w1(X) conflict, which means that we cannot move r2(X) down to get the equivalent serial schedule T1, T2. Similarly, because w1(X) and w2(X) conflict, we cannot move w1(X) down to get the equivalent serial schedule T2, T1.

How Serializability Is Used for Concurrency Control

A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is quite difficult to test for the serializability of a schedule. Hence, the approach taken in most practical systems is to determine methods or protocols that ensure serializability, without having to test the schedules themselves. The approach taken in most commercial DBMSs is to

design protocols (sets of rules) that—if followed by every individual transaction or if enforced by a DBMS concurrency control subsystem—will ensure serializability of all schedules in which the transactions participate.

Conflict Serializability

Instructions Ii and Ij of transactions Ti and Tj respectively, conflict if and only if there exists some item Q accessed by both Ii and Ij, and at least one of these instructions wrote Q.

1. If I_i and I_j access different data item then I_i and I_j don't conflict.
2. I_i = read(Q), I_j = read(Q). I_i and I_j don't conflict.
3. I_i = read(Q), I_j = write(Q). I_i and I_j conflict.
4. I_i = write(Q), I_j = read(Q). I_i and I_j conflict.
5. I_i = write(Q), I_j = write(Q). I_i and I_j conflict.

Intuitively, a conflict between Ii and Ij forces a (logical) temporal order between them. If a schedule S can be transformed into a schedule S´ by a series of swaps of non-conflicting instructions, we say that S and S´ are conflict equivalent. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

As shown in Fig. Schedule S can be transformed into Schedule S' by swapping of non-conflicting series of instructions. Therefore Schedule S is conflict serializable.

Schedule S	
T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule S'	
T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

In above example the write(A) instruction of transaction T1 conflict with read(A) instruction of transaction T2 because both the instructions access same data A. But write(A) instruction of transaction T2 is not conflict with read(B) instruction of transaction T1 because both the instructions access different data. Transaction T2 performs write operation in A and transaction T1 is reading B. So in above example in schedule S two instructions read(A) and write(A) of transaction T2 and two instructions read(B) and write(B) of transaction T1 are interchanged and we get schedule S'. Therefore Schedule S is conflict serializable.

Schedule S''	
T3	T4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule S'' to obtain either the serial schedule < T3, T4 >, or the serial schedule < T4, T3 >. So above schedule S'' is not conflict serializable.

View Equivalence and View Serializability

Other than conflict equivalence there is another less restrictive definition of equivalence of schedules called *view equivalence*. This leads to another definition of serializability called *view serializability*. Two schedules S and S' 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 $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (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 $r_i(X)$ of T_i in S'.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S'.

The idea behind view equivalence is that, 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 read operations are hence said to see the same view in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules. A schedule S is said to be *view serializable* if it is view equivalent to a serial schedule.

View serializability

A schedule S is view serializable if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable but every view serializable is not conflict serializable. Below is a schedule which is view serializable but not conflict serializable.

Schedule S		
T3	T4	T6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

In S the operations T4: write(Q) and T6: write(Q) are blind writes, since T4 and T6 do not read the value of X. The schedule S is view serializable, since it is view equivalent to the serial schedule T3, T4, T6. However, S is not conflict serializable, since it is not conflict equivalent to any serial schedule. It has been shown that any conflict serializable schedule is also view serializable but not vice versa, as illustrated by the preceding example.

CONCURRENCY CONTROL

LOCKS

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A *lock* is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted. One major problem in databases is concurrency. Concurrency problems arise when multiple users try to update or insert data into a database table at the same time. Such concurrent updates can cause data to become corrupt or inconsistent. Locking is a strategy that is used to prevent such concurrent updates to data. A lock is a mechanism to control concurrent access to a data item. Data items can be locked in two modes :

1. **Exclusive-lock (X) / write_Lock(X):** Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. **Shared_lock / read_lock(X):** Data item can only be read. S-lock is requested using lock-S instruction.

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions. Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no

other transaction may hold any lock on the item. If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted. When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed, as we discuss shortly.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed, as we discuss shortly.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks

Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow lock conversion; that is, a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to upgrade the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to downgrade the lock by issuing a read_lock(X) operation.

TWO PHASE LOCKING PROTOCOL

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases: an *expanding or growing (first) phase*, during which new locks on items can be acquired but none can be released; and a *shrinking (second) phase*, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a

read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

T_1	T_2

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 of schedule S:
X=50, Y=50
(nonserializable)

Figure 22.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

Transactions T1 and T2 in Figure (a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2. If we enforce two-phase locking, the transactions can be rewritten as T1' and T2', as shown in Figure below. Now, the schedule shown in Figure (c) is not permitted for T1' and T2' (with their modified order of locking and unlocking operations) under the rules of locking, because T1' will issue its write_lock(X) before it unlocks item Y; consequently, when T2' issues its read_lock(X), it is forced to wait until T1' releases the lock by issuing an unlock (X) in the schedule.

T_1'	T_2'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	write_lock(Y);
	unlock(X)
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

Figure
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit all possible serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

Timestamp Based Concurrency Control

A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Timestamps

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time. We will refer to the timestamp of transaction T as TS(T). Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. This is called timestamp ordering (TO). The algorithm must ensure that, for each item accessed by conflicting operations in the schedule, the order in which the item is accessed does not violate the timestamp order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. **read_TS(X):** The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the youngest transaction that has read X successfully.
2. **write_TS(X):** The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the youngest transaction that has written X successfully.

Basic Timestamp Ordering (TO)

Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the basic TO algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp. If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1) Whenever a transaction T issues a write_item(X) operation, the following is checked:

- **a.** If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
- **b.** If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2) Whenever a transaction T issues a read_item(X) operation, the following is checked:

- **a.** If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of item X before T had a chance to read X.

b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Whenever the basic TO algorithm detects two conflicting operations that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be conflict serializable. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO)

A variation of basic TO called strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction T' that wrote the value of X (hence TS(T') = write_TS(X)) has committed or aborted. To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction Tuntil T is either committed or aborted. This algorithm does not cause deadlock, since T waits for T' only if TS(T) > TS(T).

RECOVERY TECHNIQUES

Recovery from transaction failures usually means that the database is restored to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the system log. A typical strategy for recovery may be summarized informally as follows:

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was backed up to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or redoing the operations of committed transactions from the backed up log, up to the time of failure.

2. When the database on disk is not physically damaged, and a non catastrophic failure of has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by undoing its write operations. It may also be necessary to redo some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For non catastrophic failure, the recovery protocol does not need a complete archival copy of the

database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main techniques for recovery from non catastrophic transaction failures: *deferred update* and *immediate update*. The deferred update techniques do not physically update the database on disk until after a transaction reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache). Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk. Hence, deferred update is also known as the NO-UNDO/REDO algorithm.

In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations must also be recorded in the log on disk by force-writing before they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both undo and redo may be required during recovery. This technique, known as the *UNDO/REDO algorithm*, requires both operations during recovery, and is used most often in practice.

Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery. In this case, the recovery mechanism must ensure that the BFIM (Before Image) of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM (After Image) in the database on disk. This process is generally known as *write-ahead logging*, and is necessary to be able to UNDO the operation if this is required during recovery.

A REDO-type log entry includes the new value (AFIM) of the item written by the operation since this is needed to redo the effect of the operation from the log (by setting the item value in the database on disk to its AFIM). The UNDO-type log entries include the old value (BFIM) of the item since this is needed to undo the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both types of log entries are combined. Additionally, when

cascading rollback is possible, read_item entries in the log are considered to be UNDO-type entries.

With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update must first be written to disk before the data block itself can be written back to disk from its main memory buffer.

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force**, which specify the rules that govern when a page from the database can be written to disk from the cache:

1. If a cache buffer page updated by a transaction cannot be written to disk before the transaction commits, the recovery method is called a no-steal approach. On the other hand, if the recovery protocol allows writing an updated buffer before the transaction commits, it is called **steal**. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The **no-steal** rule means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.

2. If all pages updated by a transaction are immediately written to disk before the transaction commits, it is called a **force** approach. Otherwise, it is called **no-force**. The force rule means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

ARIES Recovery Algorithm

ARIES uses a steal/no-force approach for writing, and it is based on three concepts: write-ahead logging, repeating history during redo, and logging changes during undo. Repeating history, means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred. Transactions that were uncommitted at the time of the crash (active transactions) are undone. The third concept, logging during undo, will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery procedure consists of three main steps: analysis, REDO, and UNDO. The analysis step identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The REDO phase actually reapplies updates from the log to the database. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. Additionally, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied

to the database and therefore does not need to be reapplied. Thus, only the necessary REDO operations are applied during recovery. Finally, during the UNDO phase, the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used. These tables are maintained by the transaction manager and written to the log during checkpointing.

In ARIES, every log record has an associated log sequence number (LSN) that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a specific change (action) of some transaction. Also, each data page will store the LSN of the latest log record corresponding to a change for that page. A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end).

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include the page ID for the page that contains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

Besides the log, two tables are needed for efficient recovery: the Transaction Table and the Dirty Page Table, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for each active transaction, with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The 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.

Checkpointing in ARIES consists of the following: writing a begin_checkpoint record to the log, writing an end_checkpoint record to the log, and writing the LSN of the begin_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the end_checkpoint record, the contents of both the Transaction Table and Dirty Page Table are appended to the end of the log. Additionally, the contents of the DBMS cache do not have to be flushed to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. If a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which is used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The analysis phase starts at the begin_checkpoint record and proceeds to the end of the log.When the end_checkpoint record is encountered, the Transaction Table and Dirty Page Table are accessed. During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction T in the Transaction Table, then the entry for T is deleted from that table. If some other type of log record is encountered for a transaction T', then an entry for T' is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page P, then an entry would be made for page P (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The REDO phase follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages have already been applied to the database on disk. It can determine this by finding the smallest LSN, M, of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to an LSN < M, for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = M and scans forward to the end of the log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page P that is not in the Dirty Page Table, then this change is already on disk and does not need to be reapplied. Or, if a change recorded in the log (with LSN = N, say) pertains to page P and the Dirty Page Table contains an entry for P with LSN greater than N, then the change is already present. If neither of these two conditions hold, page P is read from disk and the LSN stored on that page, LSN(P), is compared with N. If N < LSN(P), then the change has been applied and the page does not need to be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the undo_set—has been identified in the Transaction Table during the analysis phase. Now, the UNDO phase proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the undo_set has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) is based on the Write Ahead Log (WAL) protocol. Every update operation writes a log record which is one of the following :

1. **Undo-only log record:**
   Only the before image is logged. Thus, an undo operation can be done to retrieve the old data.
2. **Redo-only log record:**
   Only the after image is logged. Thus, a redo operation can be attempted.
3. **Undo-redo log record:**
   Both before images and after images are logged.

In it, every log record is assigned a unique and monotonically increasing log sequence number (LSN). Every data page has a page LSN field that is set to the LSN of the log record corresponding to the last update on the page. WAL requires that the log record corresponding to an update make it to stable

storage before the data page corresponding to that update is written to disk. For performance reasons, each log write is not immediately forced to disk. A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full. A transaction cannot be declared committed until the commit log record makes it to disk.

Once in a while the recovery subsystem writes a checkpoint record to the log. The checkpoint record contains the transaction table and the dirty page table. A master log record is maintained separately, in stable storage, to store the LSN of the latest checkpoint record that made it to disk. On restart, the recovery subsystem reads the master log record to find the checkpoint's LSN, reads the checkpoint record, and starts recovery from there on.

The recovery process actually consists of 3 phases:

1. **Analysis:**
   The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.
2. **Redo:**
   Starting at the earliest LSN, the log is read forward and each update redone.
3. **Undo:**
   The log is scanned backward and updates corresponding to loser transactions are undone.

Distributed Database – Concurrency Control

In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

Two Phase Commit Protocol

The two phase commit protocol provides an automatic recovery mechanism in case a system or media failure occurs during execution of the transaction. The

two phase commit protocol ensures that all participants perform the same action (either to commit or to roll back a transaction). The two phase commit strategy is designed to ensure that either all the databases are updated or none of them, so that the databases remain synchronized.

In two phase commit protocol there is one node which is act as a coordinator and all other participating node are known as cohorts or participant.

Coordinator – the component that coordinates with all the participants.

Cohorts (Participants) – each individual node except coordinator are participant.

As the name suggests, the two phase commit protocol involves two phases. The first phase is *Commit Request* phase OR phase 1. The second phase is *Commit phase* OR phase 2.

Commit Request Phase (Obtaining Decision)

To commit the transaction, the coordinator sends a request asking for "ready for commit" to each cohort. The coordinator waits until it has received a reply from all cohorts to "vote" on the request. Each participant votes by sending a message back to the coordinator as follows:

a) It votes YES if it is prepared to commit

b) It may vote NO for any reason if it cannot prepare the transaction due to a local failure.

c) It may delay in voting because cohort was busy with other work.

Commit Phase (Performing Decision)

If the coordinator receives YES response from all cohorts, it decides to commit. The transaction is now officially committed. Otherwise, it either receives a NO response or gives up waiting for some cohort, so it decides to abort. The coordinator sends its decision to all participants (i.e. COMMIT or ABORT). Participants acknowledge receipt of commit or about by replying DONE.