Chapter 1 Introduction:

1.1 Concept and Applications:

Data: Collection of numbers, characters/ Data are values of qualitative or quantitative variables, belonging to a set of items.

Database: A database is an organized collection of data. The data are typically organized to model relevant aspects of reality in a way that supports processing requiring this information. For example, modeling the availability of rooms in hotels in a way that supports finding a hotel with vacancies.

Database Management System: A Database Management System is a collection of interrelated data and a set of programs to access those data. The main purpose of DBMS is to provide a way to store and retrieve database information that is both convenient and efficient. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access.

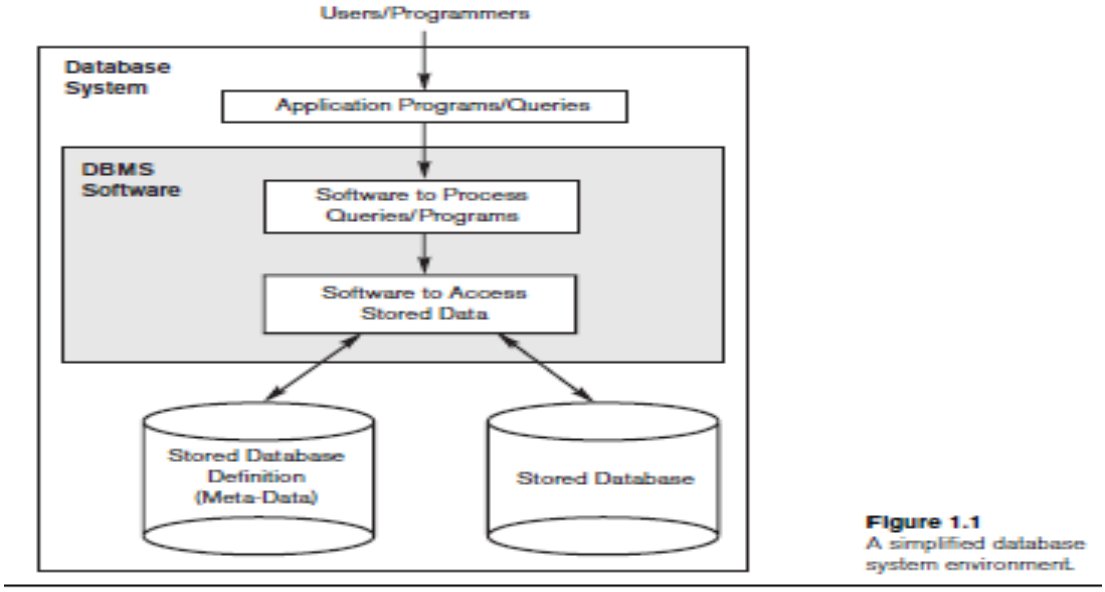


Figure 1.1
A simplified database system environment.

Applications of Database System:

Databases are widely used. Here are some representative applications:
• *Banking*: For customer information, accounts, and loans, and banking transactions.
• *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.
• *Universities*: For student information, course registrations, and grades.
• *Credit card transactions*: For purchases on credit cards and generation of monthly statements.

• *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
• *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.
• *Sales*: For customer, product, and purchase information.
• *Manufacturing*: For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.
• *Human resources*: For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

1.2 Objectives of DBMS:

The objectives that the management should keep in mind when they design and organize their data base management systems are:
(i) Provide for mass storage of relevant data,
(ii) Make access to the data easy for the user,
(iii) Provide prompt response to user requests for data,
(iv) Make the latest modifications to the database available immediately,
(v) Eliminate redundant data,
(vi) Allow for multiple users to be active at one time,
(vii) Allow for growth in the database system,
(viii) Protect the data from physical harm and unauthorised access.

Evolution of DBMS:

- Over the course of the last four decades of the twentieth century, use of databases grew in all enterprises. In the early days, very few people interacted directly with database systems, although without realizing it they interacted with databases indirectly— through printed reports such as credit card statements, or through agents such as bank tellers and airline reservation agents.
- Then automated teller machines came along and let users interact directly with databases. Phone interfaces to computers (interactive voice response systems) also allowed users to deal directly with databases—a caller could dial a number, and press phone keys to enter information or to select alternative options, to find flight arrival/departure times, for example, or to register for courses in a university.
- The internet revolution of the late 1990s sharply increased direct user access to databases. Organizations converted many of their phone interfaces to databases into Web interfaces, and made a variety of services and information available online. For instance, when you access an online bookstore and browse a book or music collection, you are accessing data stored in a database.
- Moreover, now there Multimedia databases which can store pictures, video chips and sound messages. Geographic Information Systems (GIS) to store and analyze maps, weather data and satellite image. Data Ware house and Online Analytic Processing (OLAP) systems are used in many companies to extract and analyze useful information for decision making.

1.3 Needs of DBMS:

a) The ability to update and retrieve data :
This is a fundamental component of a DBMS and essential to database management. Without the ability to view or manipulate data, there would be no point to using a database system.

Updating data in a database includes adding new records, deleting existing records and changing information within a record. The user does not need to be aware of how DBMS structures this data, all the user needs to be aware of is the availability of updating and/or pulling up information, the DBMS handles the processes and the structure of the data on a disk.

b) Support Concurrent Updates :
Concurrent updates occur when multiple users make updates to the database simultaneously. Supporting concurrent updates is also crucial to database management as this component ensures that updates are made correctly and the end result is accurate. Without DBMS intervention, important data could be lost and/or inaccurate data stored.
DBMS uses features to support concurrent updates such as batch processing, locking, two-phase locking, and time stamping to help make certain that updates are done accurately. Again, the user is not aware all this is happening as it is the database management system's responsibility to make sure all updates are stored properly.

c) Recovery of Data :
In the event a catastrophe occurs, DBMS must provide ways to recover a database so that data is not permanently lost. There are times computers may crash, a fire or other natural disaster may occur, or a user may enter incorrect information invalidating or making records inconsistent.
If the database is destroyed or damaged in any way, the DBMS must be able to recover the correct state of the database, and this process is called Recovery. The easiest way to do this is to make regular backups of information. This can be done at a set structured time so in the event a disaster occurs, the database can be restored to the state that it was last at prior to backup

1.4 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-systems users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:
• **Physical level**. The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.
• **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
• **View level**. 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 1.1 shows the relationship among the three levels of abstraction.

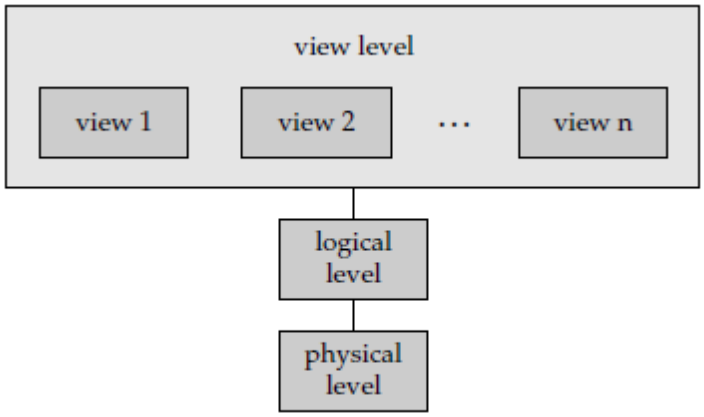


Figure 1.1 The three levels of data abstraction.

1.5 Data Independence:

IT can be defined as the capacity to change the schema at one level of a database system without having to change schema at the next higher level. There are mainly two types of Data Independence

1. Logical data independence: The ability to change the logical (conceptual) schema without changing the External schema (User View) is called logical data independence. For example, the addition or removal of new entities, attributes, or relationships to the conceptual schema should be possible without having to change existing external schemas or having to rewrite existing application programs.
2. Physical data independence: The ability to change the physical schema without changing the logical schema is called physical data independence. For example, a change to the internal schema, such as using different file organization or storage structures, storage devices, or indexing strategy, should be possible without having to change the conceptual or external schemas.

1.6 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. The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. 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**, that describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

1.7 Database Languages(Concept of DDL, DML, DCL)

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates.

1.7.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**).

For instance, the following statement in the SQL language defines the *account* table:

create table *account* (*account-number* **char**(10), *balance* **integer**)

Execution of the above DDL statement creates the *account* table. In addition, it updates a special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains **metadata**—that is, data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading or modifying actual data.

1.7.2 Data-Manipulation Language

Data manipulation is

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database
- The modification of information stored in the database

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. There are basically two types:

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

The DML component of the SQL language is nonprocedural.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

This query in the SQL language finds the name of the customer whose customer-id is 192-83-7465:

select *customer.customer-name*
from *customer*
where *customer.customer-id* = 192-83-7465

1.7.3 Data Control Language(DCL): DCL languages are used to control the user access to the database, tables, views, procedures, functions and packages. They give different levels of access to the objects in the database. Some Examples:

Grant- gives user's access privileges to database.
Revoke- Withdraw access privileges given with GRANT command.

1.8 Database Users and Administrators

People who work with a database can be categorized as database users or database administrators.

1.8.1 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 who needs to transfer $50 from account A to account B invokes a program called transfer. This program asks the teller for the amount of

money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred. 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. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program.

• **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. **Online analytical processing (OLAP)** tools simplify analysts' tasks by letting them view summaries of data in different ways.

• **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.

1.8.2 Database Administrator(Manager)

A person having the central control over the system is called a **database administrator** (**DBA**) or **database manager**. The functions of a DBA include:

- **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition**.
- **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
  - Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
  - Ensuring that enough free disk space is available for normal operations, and upgrading disk Space as required.
  - Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

Chapter 2: Data Models

2.1 Conceptual, Logical and Physical Model:

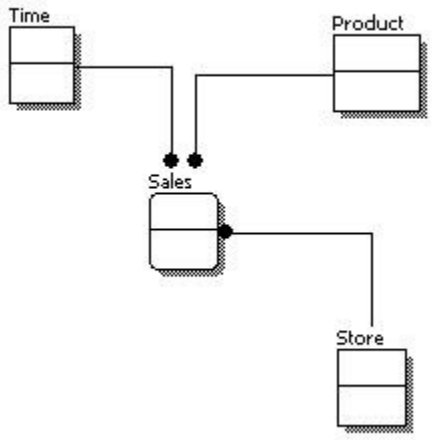
Conceptual Data Model:

A conceptual data model identifies the highest-level relationships between the different entities.
Features of conceptual data model include:

- Includes the important entities and the relationships among them.
- No attribute is specified.
- No primary key is specified.

The figure below is an example of a conceptual data model.

Conceptual Data Model



From the figure above, we can see that the only information shown via the conceptual data model is the entities that describe the data and the relationships between those entities. No other information is shown through the conceptual data model.

Logical Data Model:

A logical data model describes the data in as much detail as possible, without regard to how they will be physical implemented in the database. Features of a logical data model include:

- Includes all entities and relationships among them.
- All attributes for each entity are specified.
- The primary key for each entity is specified.
- Foreign keys (keys identifying the relationship between different entities) are specified.
- Normalization occurs at this level.

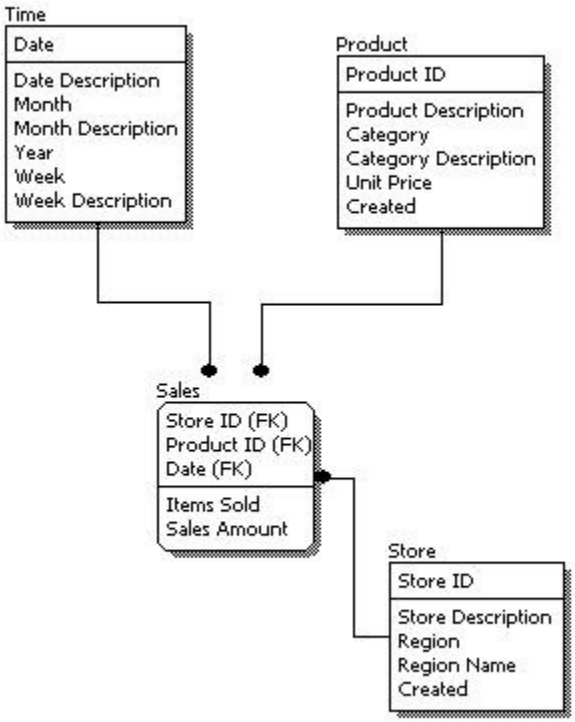
The steps for designing the logical data model are as follows:

1. Specify primary keys for all entities.

2. Find the relationships between different entities.
3. Find all attributes for each entity.
4. Resolve many-to-many relationships.
5. Normalization.

The figure below is an example of a logical data model.

Logical Data Model



Comparing the logical data model shown above with the conceptual data model diagram, we see the main differences between the two:

- In a logical data model, primary keys are present, whereas in a conceptual data model, no primary key is present.
- In a logical data model, all attributes are specified within an entity. No attributes are specified in a conceptual data model.
- Relationships between entities are specified using primary keys and foreign keys in a logical data model. In a conceptual data model, the relationships are simply stated, not specified, so we simply know that two entities are related, but we do not specify what attributes are used for this relationship.

Physical Data Model:

Physical data model represents how the model will be built in the database. A physical database model shows all table structures, including column name, column data type, column constraints, primary key, foreign key, and relationships between tables. Features of a physical data model include:

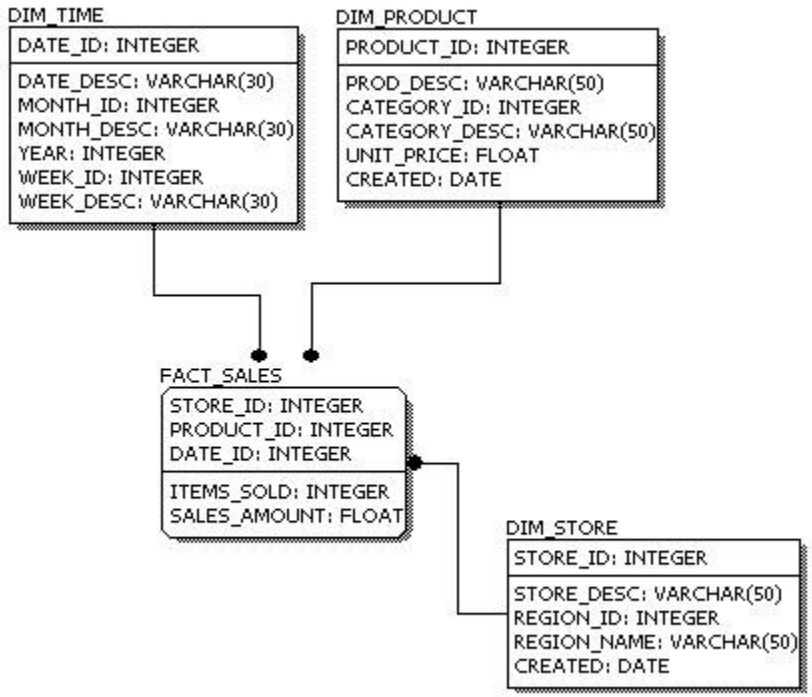
- Specification all tables and columns.
- Foreign keys are used to identify relationships between tables.
- Denormalization may occur based on user requirements.
- Physical considerations may cause the physical data model to be quite different from the logical data model.
- Physical data model will be different for different RDBMS. For example, data type for a column may be different between MySQL and SQL Server.

The steps for physical data model design are as follows:

1. Convert entities into tables.
2. Convert relationships into foreign keys.
3. Convert attributes into columns.
4. Modify the physical data model based on physical constraints / requirements.

The figure below is an example of a physical data model.

Physical Data Model



Comparing the physical data model shown above with the logical data model diagram, we see the main differences between the two:

- Entity names are now table names.
- Attributes are now column names.
- Data type for each column is specified. Data types can be different depending on the actual database being used.

The table below compares the different features:

Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

2.2 E-R Model:

The entity-relationship(E-R) data model is based on a perception of a real world that consists of a collection of a basic objects, called entities, and of relationships among these objects. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. It employs three basic notions: entity sets, relationship sets, and attributes.

- **Entities and Their Attributes.** The basic object that the ER model represents is an **entity**, which is a *thing* in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course). Each entity has **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.
- A **relationship** is an association among several entities. For example, a **depositor** relationship associates a customer with each account they have. The set of all entities of the same type and the set of all the relationships of the same type are termed as **entity set** and **relationship set** respectively.

Types Of Attributes:

• **Simple** and **composite** attributes. The attributes are not divided into subparts are called **Simple** attributes. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first-name*, *middle-initial*, and *last-name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Figure 2.2 depicts these examples of composite attributes for the *customer* entity set.

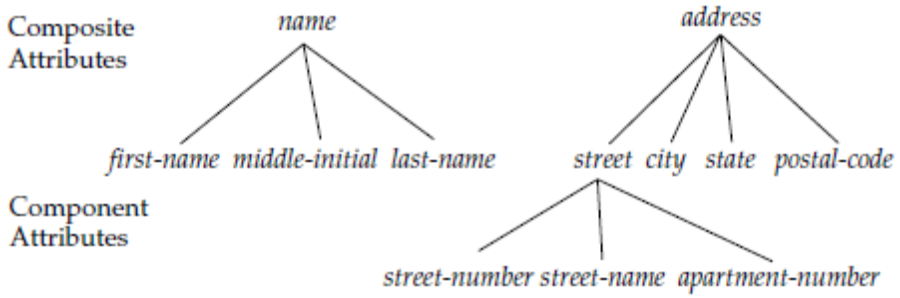


Figure 2.2 Composite attributes *customer-name* and *customer-address*.

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

- **Derived** attribute. The value for this type of attribute can be derived from the values of other related attributes or entities. For example, If the *customer* entity set also has an attribute *date-of-birth*, we can calculate *age* from *date-of-birth* and the current date. Thus, *age* is a derived attribute. In this case, *date-of-birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored, but is computed when required.
- An attribute takes a **null** value when an entity does not have a value for it. The *null* value may indicate "not applicable"—that is, that the value does not exist for the entity. For example, one may have no middle name. *Null* can also designate that an attribute value is unknown. An unknown value may be either *missing* (the value does exist, but we do not have that information) or *not known* (we do not know whether or not the value actually exists).

Constraints:

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

Mapping Cardinalities:

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.

For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:

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

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

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

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

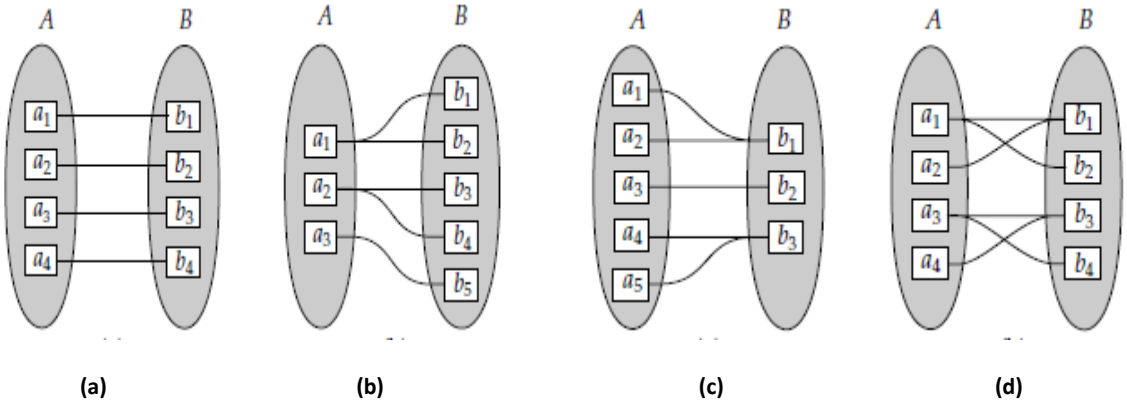


Fig: Mapping Cardinalities (a) One to one. (b)One to many. (c) Many to one. (d) Many to many

Participation Constraints:

The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*. If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**. For example, we expect every loan entity to be related to at least one customer through the *borrower* relationship. Therefore the participation of *loan* in the relationship set *borrower* is total. In contrast, an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the *customer* entities are related to the *loan* entity set through the *borrower* relationship, and the participation of *customer* in the *borrower* relationship set is therefore partial.

Entity-Relationship Diagram:

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

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

Consider the entity-relationship diagram in Figure below, which consists of two entity sets, *customer* and *loan*, related through a binary relationship set *borrower*. The attributes associated with *customer* are *customer-id*, *customer-name*, *customer-street*, and *customer-city*. The attributes associated with *loan* are *loan-number* and *amount*. In Figure, attributes of an entity set that are members of the primary key are underlined. The relationship set *borrower* may be many-to-many, one-to-many, many-to-one, or one-to-one. To distinguish among these types, we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question.

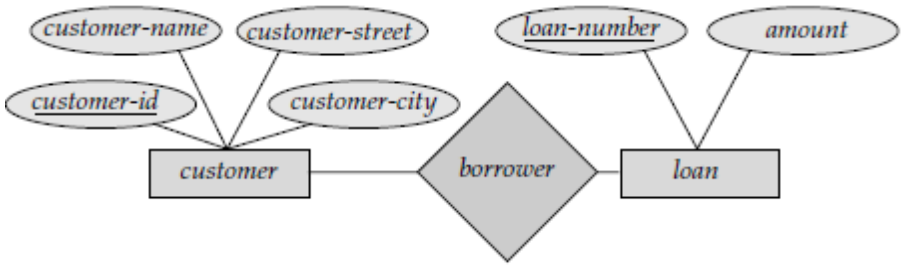


Fig: E-R Diagram corresponding to Customers and loans

• A directed line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a one-to-one or many-to-one relationship set, from *customer* to *loan*; *borrower* cannot be a many-to-many or a one-to-many relationship set from *customer* to *loan*.
• An undirected line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a many-to-many or one-to-many relationship set from *customer* to *loan*.
Returning to the E-R diagram of above Figure , we see that the relationship set *borrower* is many-to-many. If the relationship set *borrower* were one-to-many, from *customer* to *loan*, then the line from *borrower* to *customer* would be directed, with an arrow pointing to the *customer* entity set . Similarly, if the relationship set *borrower* were many-to-one from *customer* to *loan*, then the line from *borrower* to *loan* would have an arrow pointing to the *loan* entity set . Finally, if the relationship set *borrower* were one-to-one, then both lines from *borrower* would have arrows: one pointing to the *loan* entity set and one pointing to the *customer* entity set .

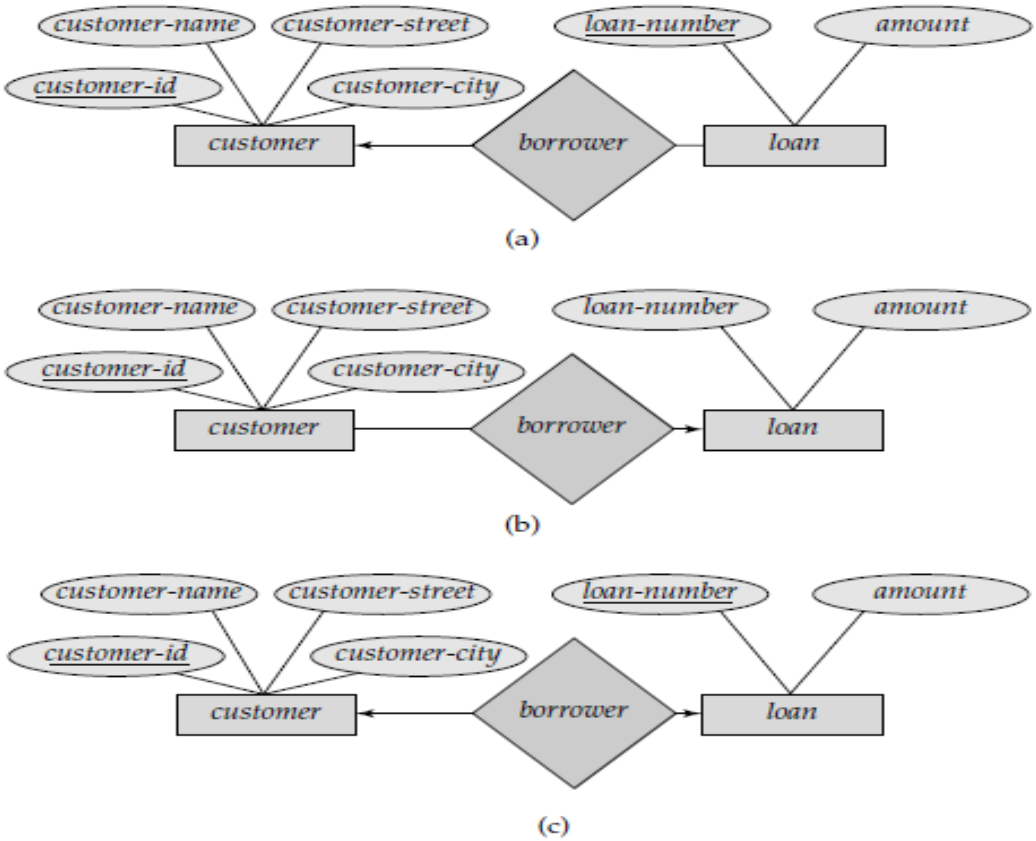


Fig: Relationships.(a) One to many.(b) Many to one.(c)One to one.

If a relationship set has also some attributes associated with it, then we link these attributes to that relationship set. For example, in Figure 2.10, we have the *access-date* descriptive attribute attached to the relationship set *depositor* to specify the most recent date on which a customer accessed that account.

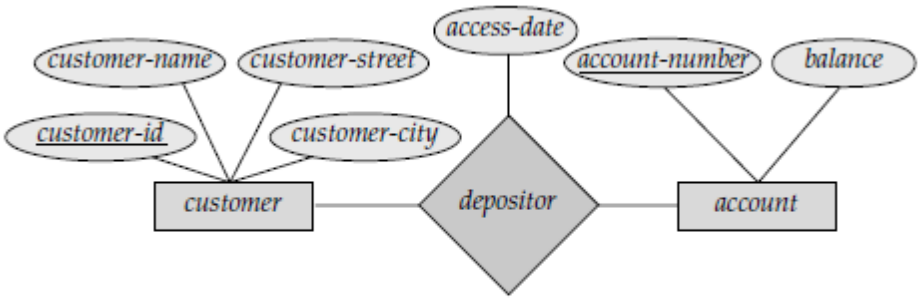


Figure 2.10 E-R diagram with an attribute attached to a relationship set.

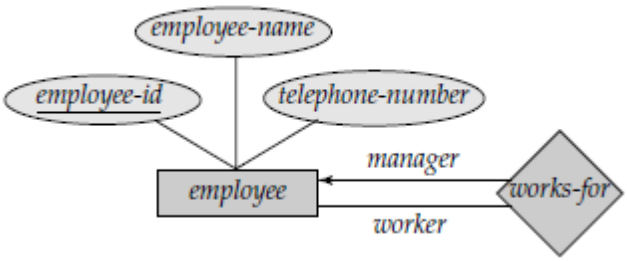


Fig: E-R diagram with role indicators.

Example: Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

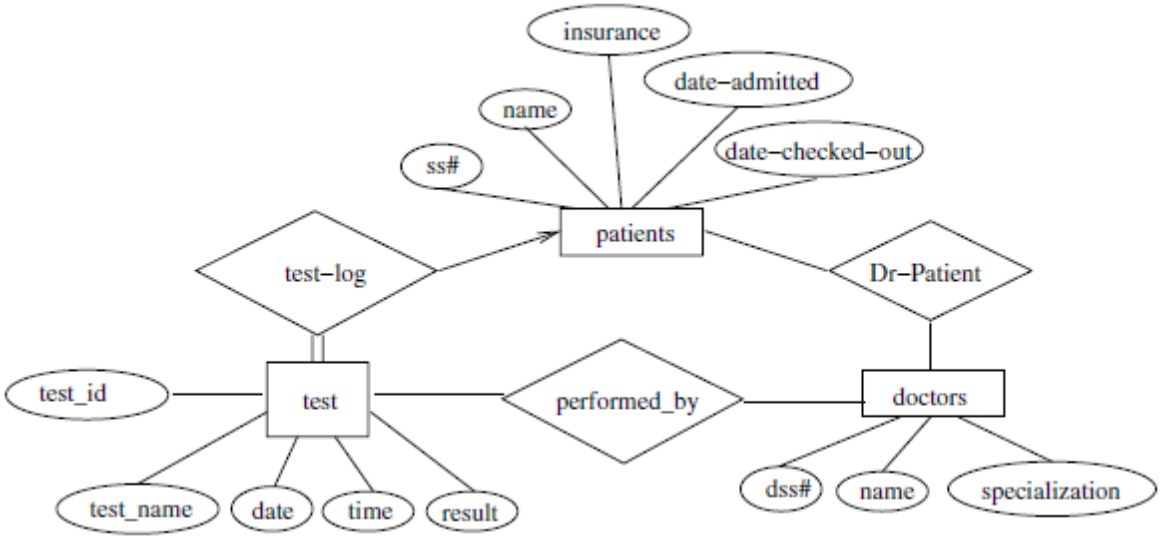


Fig: E-R diagram for a hospital

Example: Consider a database to record the marks that students get in different exams of different course offerings. Construct an E-R diagram that shows relationship, for the above database.

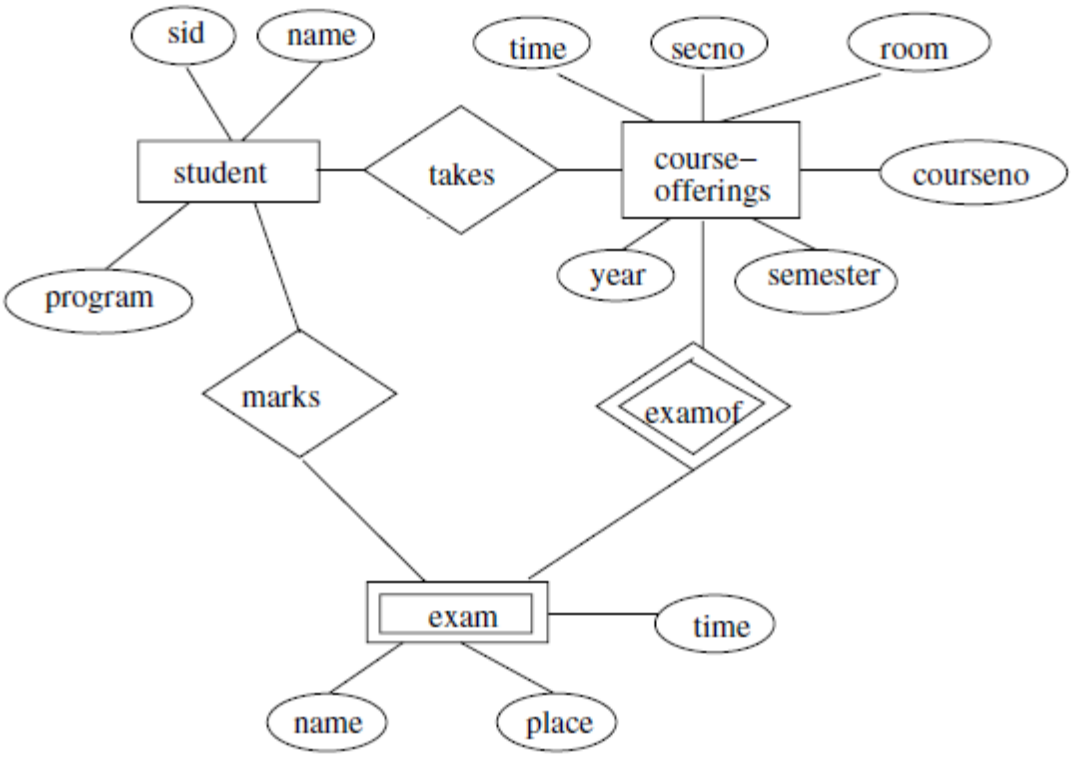


Fig: E-R diagram for marks database

Extended E-R Features:

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. Here, we discuss the extended E-R features of specialization and generalization.

Specialization:

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings. Consider an entity set *person*, with attributes *name*, *street*, and *city*. A person may be further classified as one of the following:

- *customer*
- *employee*

Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For example, *customer* entities may be described further by the attribute *customer-id*, whereas *employee* entities may be described further by the attributes *employee-id* and *salary*. The process of designating subgroupings within an entity set is called **specialization**. The specialization of *person* allows us to distinguish among persons according to whether they are employees or customers.

Generalization:

The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the

basis of common features. The database designer may have first identified a *customer* entity set with the attributes *name*, *street*, *city*, and *customer-id*, and an *employee* entity set with the attributes *name*, *street*, *city*, *employee-id*, and *salary*. There are similarities between the *customer* entity set and the *employee* entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *person* is the higher-level entity set and *customer* and *employee* are lower-level entity sets. Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *customer* and *employee* subclasses. For all practical purposes, generalization is a simple inversion of specialization.

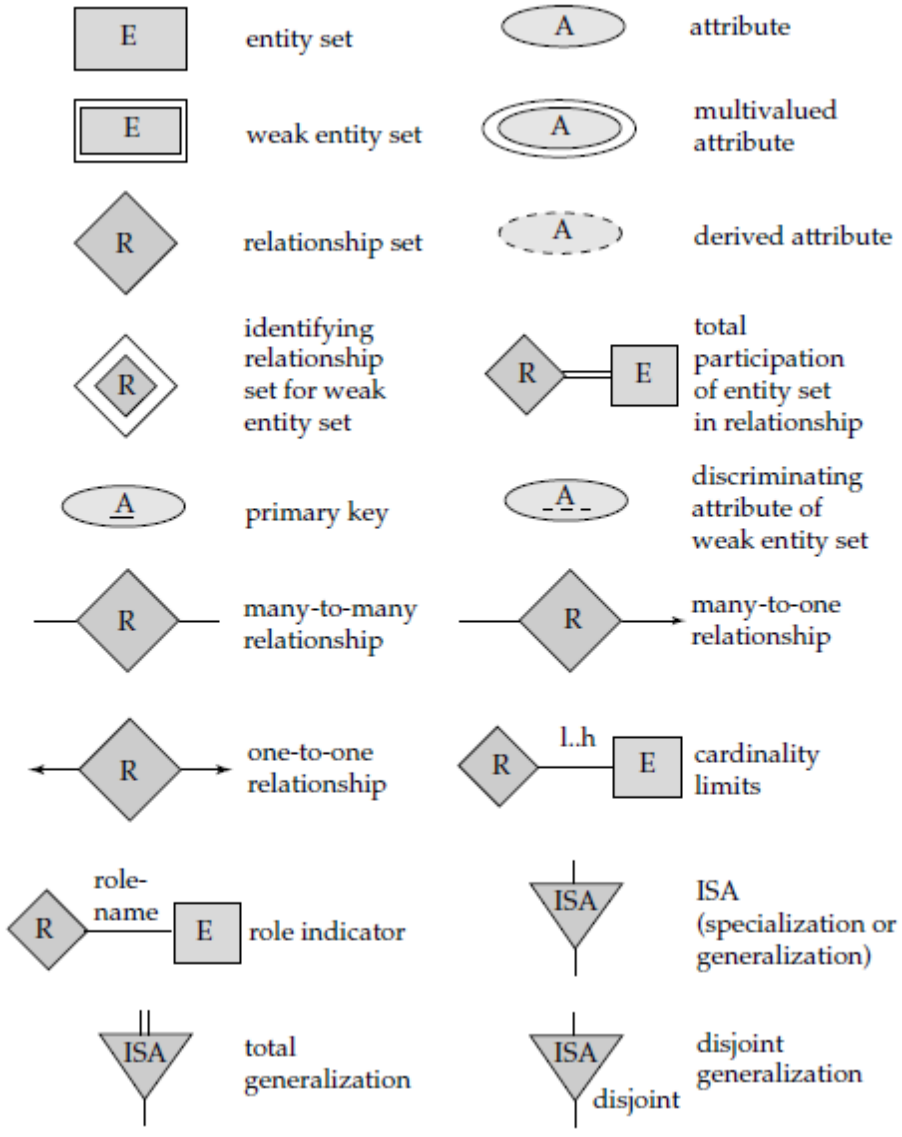


Fig: Symbols used in E-R notations

2.3 Relation with UML class diagram:

Entity-relationship diagrams help model the data representation component of a software system. Data representation, however, forms only one part of an overall system design. Other components include

models of user interactions with the system, specification of functional modules of the system and their interaction, etc. The **Unified Modeling Language** (UML), is a proposed standard for creating specifications of various components of a software system. Some of the parts of UML are:

• **Class diagram**. A class diagram is similar to an E-R diagram. Later we illustrate a few features of class diagrams and how they relate to E-R diagrams.

• **Use case diagram**. Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).

• **Activity diagram**. Activity diagrams depict the flow of tasks between various components of a system.

• **Implementation diagram**. Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

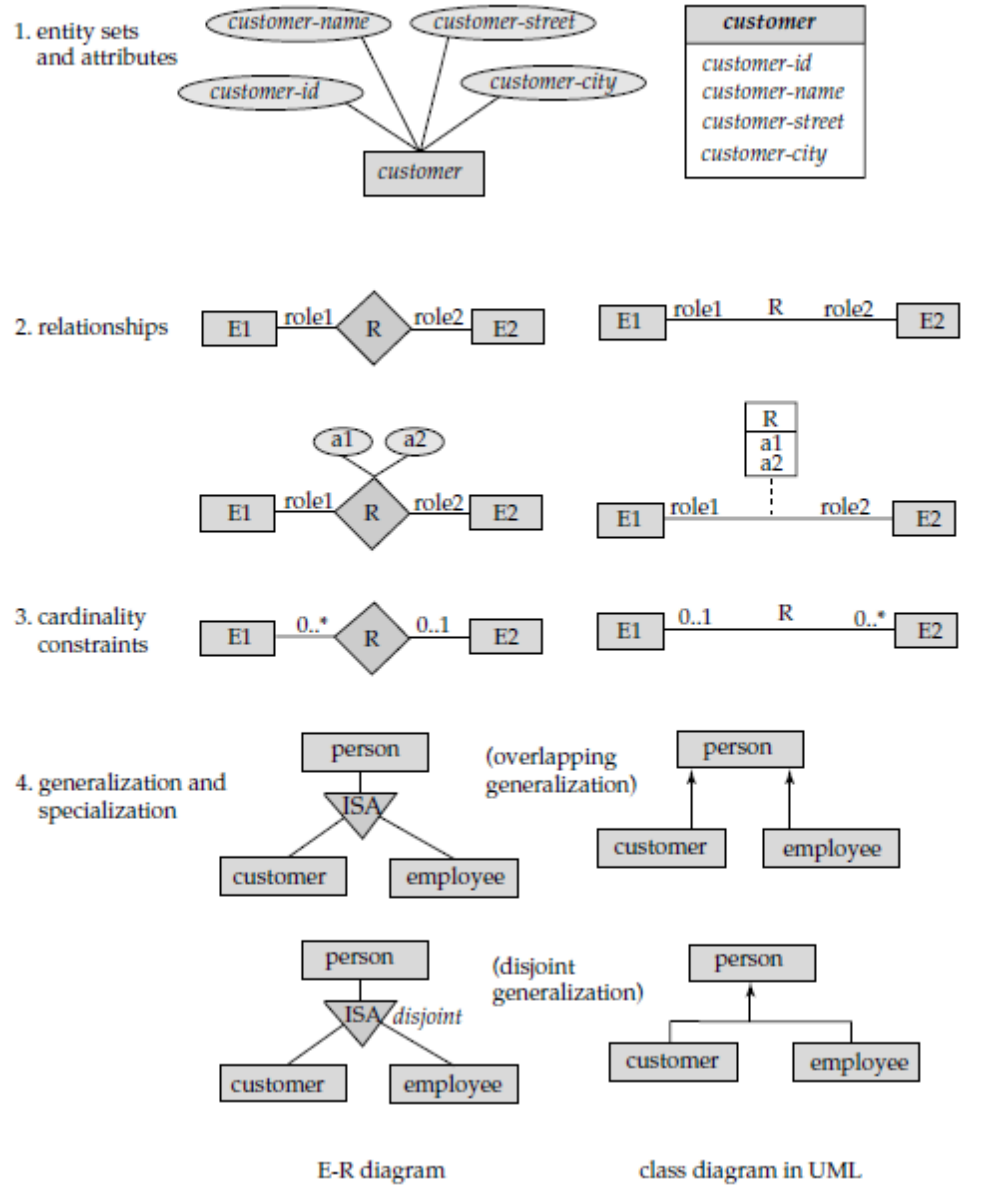
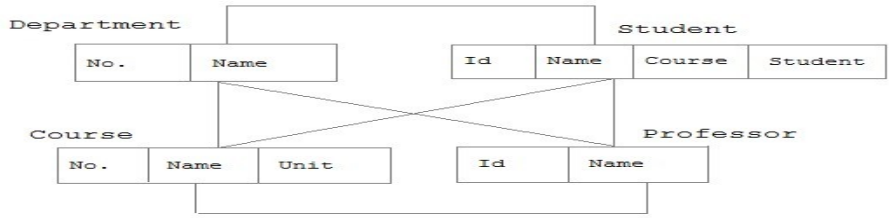


Fig: Symbols used in the UML class diagram notation

2.4 Alternative data models(Network Data Model, Heirarchical Data Model):

Network Data Model:

The **network model** is a database model conceived as a flexible way of representing objects and their relationships. Its distinguishing feature is that the schema, viewed as a graph in which object types are nodes and relationship types are arcs, is not restricted to being a hierarchy or lattice. The **network model** replaces the hierarchical model with a graph thus allowing more general connections among the nodes.The main difference of the **network model** from the hierarchical model is its ability to handle many to many relationships.In other words it allow a record to have more than one parent.
Example:



Advantages of a Network Database Model

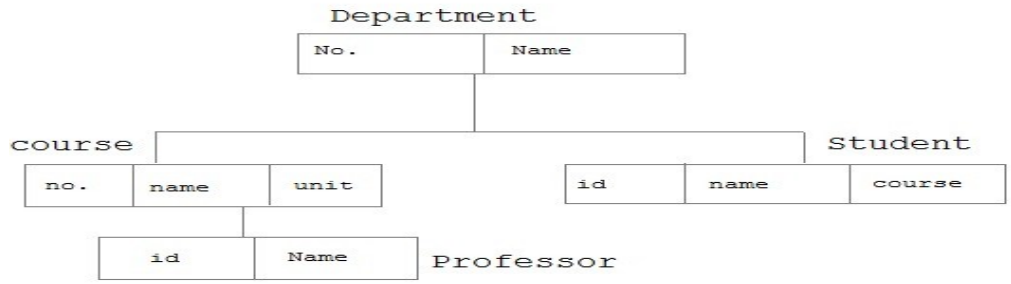
- Because it has the many-many relationship, network database model can easily be accessed in any table record in the database
- For more complex data, it is easier to use because of the multiple relationship founded among its data
- Easier to navigate and search for information because of its flexibility

Disadvantage of a Network Database Model

- Difficult for first time users
- Difficulties with alterations of the database because when information entered can alter the entire database

Hierarchical Database Model:

The hierarchical data model organizes data in a tree structure. There is a hierarchy of parent and child data segments. This structure implies that a record can have repeating information, generally in the child data segments. Data in a series of records, which have a set of field values attached to it. It collects all the instances of a specific record together as a record type. These record types are the equivalent of tables in the relational model, and with the individual records being the equivalent of rows. To create links between these record types, the hierarchical model uses Parent Child Relationships. These are a 1:N mapping between record types. This is done by using trees, like set theory used in the relational model, "borrowed" from maths. At the top of hierarchy there is only one entity which is called **Root**.
Example:



Appendix: Some Solved example of ER-Diagram

Q. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

Answer:

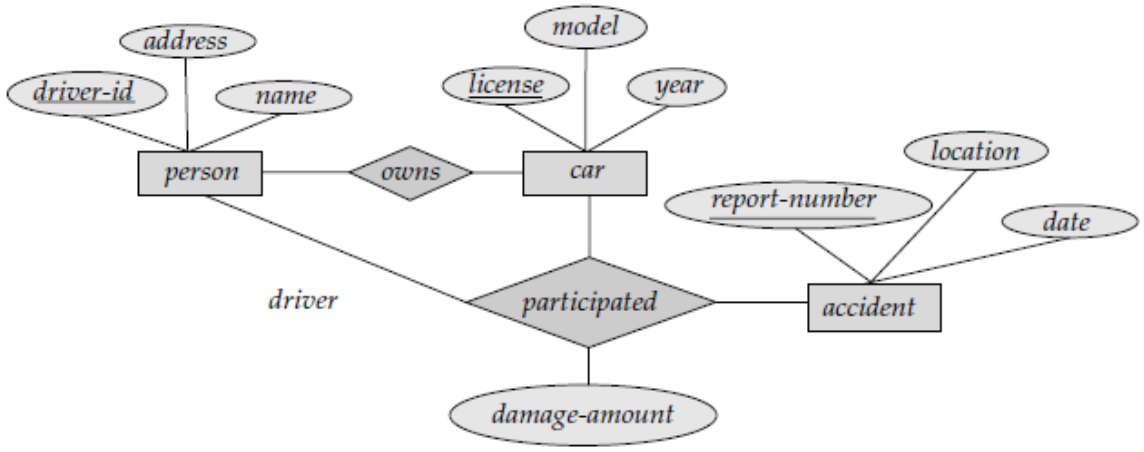


Fig: ER diagram for a car-insurance company

Q. A university registrar's office maintains data about the following entities: (a) courses, including number, title, credits, syllabus, and prerequisites; (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom; (c) students, including student-id, name, and program; and (d) instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.

Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.

Answer: Here, the main entity sets are *student, course, course-offering*, and *instructor*. The entity set *course-offering* is a weak entity set dependent on *course*. The assumptions made are :

a. a class meets only at one particular place and time. This E-R diagram cannot model a class meeting at different places at different times.

b. There is no guarantee that the database does not have two classes meeting at the same place and time.

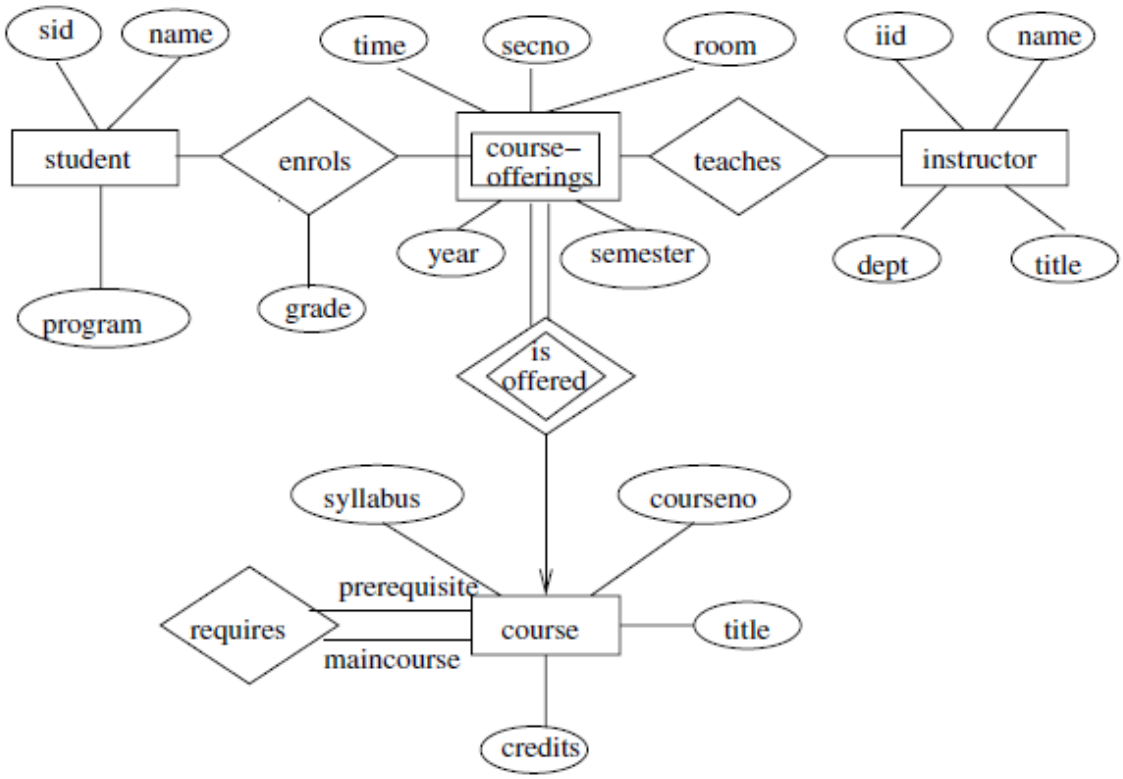


Fig: ER diagram for a university

Chapter 3: Relational Model

3.1 Schema Diagram

A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by **schema diagram**s. Figure 3.9 shows the schema diagram for our banking enterprise. Each relation appears as a box, with the attributes listed inside it and the relation name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

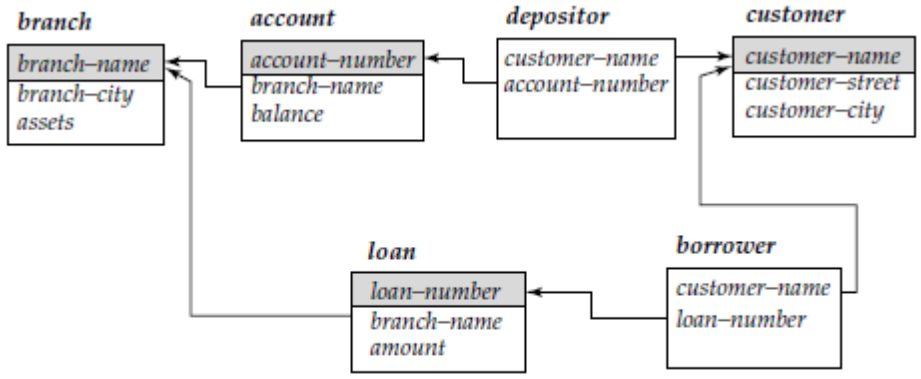


Figure 3.9 Schema diagram for the banking enterprise.

Keys in Relational Model:

Student:

Stud_id	Name	Phone_no	Address
1	Ram	9899967000	Ktm
2	Ram	9796582000	Ktm
3	suresh	9432516782	pokhara

Student_course:

Stud_no	Course_no	Course_name
1	C1	DBMS
2	C2	Computer Network
1	C1	Computer Network

Super Key: The set of one or more attributes which can uniquely identify a tuple is known as Super Key. For Example, STUD_NO, (STUD_NO, STUD_NAME) etc.

- Adding zero or more attributes to candidate key generates super key.
- A candidate key is a super key but vice versa is not true

Candidate Key: The minimal set of attribute which can uniquely identify a tuple is known as candidate key. OR A super key with no redundant attribute is known as candidate key. For Example, STUD_NO in STUDENT relation.

- The value of Candidate Key is unique and non-null for every tuple.
- There can be more than one candidate key in a relation. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT.

- The candidate key can be simple (having only one attribute) or composite as well. For Example, {STUD_NO, COURSE_NO} is a composite candidate key for relation STUDENT_COURSE.

Primary Key: There can be more than one candidate key in a relation out of which one can be chosen as primary key. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT but STUD_NO can be chosen as primary key (only one out of many candidate keys).

Foreign Key: If an attribute can only take the values which are present as values of some other attribute, it will be foreign key to the attribute to which it refers. The relation which is being referenced is called referenced relation and corresponding attribute is called referenced attribute and the relation which refers to referenced relation is called referencing relation and corresponding attribute is called referencing attribute. Referenced attribute of referencing attribute should be primary key. For Example, STUD_NO in STUDENT_COURSE is a foreign key to STUD_NO in STUDENT relation.

3.2 The Relational Algebra

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

3.2.1 Fundamental Operations

The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary o*perations.

3.2.1.1 The Select Operation

The select operation selects tuples that satisfy a given predicate.We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the *loan* relation where the branch is "Perryridge," we write

$\sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{loan})$

If the *loan* relation is as shown in Figure 3.6, then the relation that results from the preceding query is as shown in Figure 3.10.

We can find all tuples in which the amount lent is more than $1200 by writing

$\sigma_{\text{amount}>1200}(\text{loan})$

In general, we allow comparisons using $=, \neq, <, \leq, >, \geq$ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Thus, to find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write

$\sigma_{\text{branch-name} = \text{"Perryridge"} \wedge \text{amount}>1200}(\text{loan})$

loan-number	branch-name	amount
L-15	Perryridge	1500
L-16	Perryridge	1300

Figure 3.10 Result of $\sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{loan})$.

3.2.1.2 The Project Operation

Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate

rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π).We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. Thus, we write the query to list all loan numbers and the amount of the loan as

Π*loan-number, amount*(*loan*)

3.2.1.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find those customers who live in Harrison." We write:

Π*customer-name* (σ*customer-city* ="Harrison" (*customer*))

3.2.1.4 The Union Operation

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the *customer* relation does not contain the information, since a customer does not need to have either an account or a loan at the bank. To answer this query, we need the information in the *depositor* relation (Figure 3.5) and in the *borrower* relation (Figure 3.7).We know how to find the names of all customers with a loan in the bank:

Π*customer-name* (*borrower*)

We also know how to find the names of all customers with an account in the bank:

Π*customer-name* (*depositor*)

To answer the query, we need the **union** of these two sets; that is, we need all customer names that appear in either or both of the two relations.We find these data by the binary operation union, denoted, as in set theory, by ∪. So the expression needed is

Π*customer-name* (*borrower*) ∪ Π*customer-name* (*depositor*)

The result relation for this query appears in Figure 3.12. Notice that there are 10 tuples in the result, even though there are seven distinct borrowers and six depositors. This apparent discrepancy occurs because Smith, Jones, and Hayes are borrowers as well as depositors. Since relations are sets, duplicate values are eliminated.

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Figure 3.12 Names of all customers who have either a loan or an account.

Therefore, for a union operation *r* ∪ *s* to be valid, we require that two conditions hold:

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

Note that *r* and *s* can be, in general, temporary relations that are the result of relational-algebra expressions.

3.2.1.5 The Set Difference Operation

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

We can find all customers of the bank who have an account but not a loan by writing

$$\Pi customer\text{-}name\ (depositor) - \Pi customer\text{-}name\ (borrower)$$

The result relation for this query appears in Figure 3.13.
As with the union operation, we must ensure that set differences are taken between *compatible* relations. Therefore, for a set difference operation $r - s$ to be valid, we require that the relations *r* and *s* be of the same arity, and that the domains of the *i*th attribute of *r* and the *i*th attribute of *s* be the same.

3.2.1.6 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations.We write the Cartesian product of relations $r1$ and $r2$ as $r1 \times r2$.

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

Now that we know the relation schema for *r* = *borrower × loan*, what tuples appear in *r*? As you may suspect, we construct a tuple of *r* out of each possible pair of tuples: one from the *borrower* relation and one from the *loan* relation. Thus, *r* is a large relation, as you can see from Figure 3.14, which includes only a portion of the tuples that make up *r*.

Assume that we have $n1$ tuples in *borrower* and $n2$ tuples in *loan*. Then, there are $n1 * n2$ ways of choosing a pair of tuples—one tuple from each relation; so there are $n1 * n2$ tuples in *r*. In particular, note that for some tuples *t* in *r*, it may be that ***t*[*borrower.loan-number*] _= *t*[*loan.loan-number*].**

Suppose that we want to find the names of all customers who have a loan at the Perryridge branch.We need the information in both the *loan* relation and the *borrower* relation to do so. If we write

$$\sigma branch\text{-}name = \text{“Perryridge”}(borrower \times loan)$$

then the result is the relation in Figure 3.15. We have a relation that pertains to only the Perryridge branch. However, the *customer-name* column may contain customers who do not have a loan at the Perryridge branch. (If you do not see why that is true, recall that the Cartesian product takes all possible pairings of one tuple from *borrower* with one tuple of *loan*.)
Since the Cartesian-product operation associates *every* tuple of *loan* with every tuple of *borrower*, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in *borrower×loan* that contains his name, and *borrower.loan-number= loan.loan-number*. So, if we write

$$\sigma borrower.loan\text{-}number = loan.loan\text{-}number(\sigma branch\text{-}name = \text{“Perryridge”}(borrower \times loan))$$

we get only those tuples of *borrower × loan* that pertain to customers who have a loan at the Perryridge branch.
Finally, since we want only *customer-name*, we do a projection:

$$\Pi customer\text{-}name\ (\sigma borrower.loan\text{-}number = loan.loan\text{-}number(\sigma branch\text{-}name = \text{“Perryridge”}\ (borrower \times loan)))$$

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

customer-name
Johnson
Lindsay
Turner

Figure 3.13 Customers with an account but no loan.

The result of this expression, shown in Figure 3.16, is the correct answer to our query

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Figure 3.14 Result of *borrower* × *loan*.

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Figure 3.15 Result of $\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)$.

customer-name
Adams
Hayes

Figure 3.16 Result of $\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)))$

3.2.1.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression E, the expression

$$\rho x\,(E)$$

returns the result of expression *E* under the name *x*.

3.2.3 Additional Operations

3.2.3.1 The Set-Intersection Operation

The first additional-relational algebra operation that we shall define is **set intersection** ($\cap$). Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write

Π*customer-name* (*borrower*) $\cap$ Π*customer-name* (*depositor*)

Note that we can rewrite any relational algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$$r \cap s = r - (r - s)$$

3.2.3.2 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product. Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount." We first form the Cartesian product of the *borrower* and *loan* relations. Then, we select those tuples that pertain to only the same *loan-number*, followed by the projection of the resulting *customer-name*, *loan-number*, and *amount*:

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan))$$

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the "join" symbol . The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

As an illustration, consider again the example "Find the names of all customers who have a loan at the bank, and find the amount of the loan."We express this query by using the natural join as follows:

$$\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie\ loan)$$

Since the schemas for *borrower* and *loan* (that is, *Borrower-schema* and *Loan-schema*) have the attribute *loan-number* in common, the natural-join operation considers only pairs of tuples that have the same value on *loan-number*. It combines each such pair of tuples into a single tuple on the union of the two schemas (that is, *customer-name,branch-name, loan-number, amount*). After performing the projection, we obtain the relation in Figure 3.21.

customer-name	*loan-number*	*amount*
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Figure 3.21 Result of $\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie\ loan)$.

We are now ready for a formal definition of the natural join. Consider two relations $r(R)$ and $s(S)$. The **natural join** of *r* and *s*, denoted by $r\ s$, is a relation on schema $R \cup S$ formally defined as follows:

$$r \bowtie\ s\ =\ \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}\ r\ \times\ s)$$

where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

• Find the names of all branches with customers who have an account in the bank and who live in Harrison.

$$\Pi_{branch\text{-}name}\ (\sigma_{customer\text{-}city = \text{"Harrison"}}\ (customer \bowtie\ account \bowtie\ depositor))$$

Find all customers who have *both* a loan and an account at the bank.

$$\Pi_{customer\text{-}name}\ (borrower \bowtie depositor)$$

3.2.3.3 The Division Operation

The **division** operation, denoted by ÷, is suited to queries that include the phrase "for all." Suppose that we wish to find all customers who have an account at *all* the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression

$r1 = \Pi$*branch-name* (σ*branch-city* ="Brooklyn" (*branch*))

We can find all (*customer-name, branch-name*) pairs for which the customer has an account at a branch by writing

$$r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account)$$

Figure 3.24 shows the result relation for this expression.Now, we need to find customers who appear in r_2 with *every* branch name in r_1. The operation that provides exactly those customers is the divide operation. We formulate the query by writing

$$\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account) \div \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))$$

A tuple *t* is in $r \div s$ if and only if both of two conditions hold:

1. *t* is in $\Pi_{R-S}(r)$

2. For every tuple t_s in *s*, there is a tuple t_r in *r* satisfying both of the following:

a. $t_r[S] = t_s[S]$

b. $t_r[R - S] = t$

3.3.2 Aggregate Functions

Aggregate functions take a collection of values and return a single value as a result. To illustrate the concept of aggregation, we shall use the *pt-works* relation in Figure 3.27, for part-time employees. Suppose that we want to find out the total sum of salaries of all the part-time employees in the bank. The relational-algebra expression for this query is:

***G*sum(*salary*)(*pt-works*)**

The symbol G is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation G signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the hyphenated string "distinct" appended to the end of the function name (for example, count-distinct). An example arises in the query "Find the number of branches appearing in the pt-works relation."

***G*count-distinct(*branch-name*)(*pt-works*)**

Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank. To do so, we need to partition the relation *pt-works* into groups based on the branch, and to apply the aggregate function on each group.

***branch-nameG*sum(*salary*)(*pt-works*)**

Going back to our earlier example, if we want to find the maximum salary for part-time employees at each branch, in addition to the sum of the salaries, we write the expression

$$branch\text{-}name\,\mathcal{G}\,sum(salary),max(salary)(pt\text{-}works)$$

3.4 Modification of the Database

3.4.1 Deletion

We express a delete request in much the same way as a query. However, instead of displaying tuples to the user, we remove the selected tuples from the database. We can delete only whole tuples; we cannot delete values on only particular attributes.
In relational algebra a deletion is expressed by

$$r \leftarrow r - E$$

where r is a relation and E is a relational-algebra query. Here are several examples of relational-algebra delete requests:

- Delete all of Smith's account records.

$$depositor \leftarrow depositor - \sigma_{customer\text{-}name = \text{“Smith”}}(depositor)$$

- Delete all loans with amount in the range 0 to 50.

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$

- Delete all accounts at branches located in Needham.

$$r1 \leftarrow \sigma_{branch\text{-}city = \text{“Needham”}}(account \bowtie branch)$$

$$r2 \leftarrow \Pi_{branch\text{-}name,\ account\text{-}number,\ balance}(r1)$$

$$account \leftarrow account - r2$$

Note that, in the final example, we simplified our expression by using assignment to temporary relations (r_1 and r_2).

3.4.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity. The relational algebra expresses an insertion by $r \leftarrow r \cup E$

where r is a relation and E is a relational-algebra expression.
Suppose that we wish to insert the fact that Smith has $1200 in account A-973 at the Perryridge branch.We write

$$account \leftarrow account \cup \{(\text{A-973, “Perryridge”, 1200})\}$$

$$depositor \leftarrow depositor \cup \{(\text{“Smith”, A-973})\}$$

3.4.3 Updating

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple.We can use the generalized-projection operator to do this task:

$$r \leftarrow \Pi_{F1, F2, \ldots, Fn}(r)$$

where each F_i is either the ith attribute of r, if the ith attribute is not updated, or, if the attribute is to be updated, F_i is an expression, involving only constants and the attributes of r, that gives the new value for the attribute.If we want to select some tuples from r and to update only them, we can use the following expression; here, P denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi_{F1, F2, \ldots, Fn}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

To illustrate the use of the update operation, suppose that interest payments are being made, and that all balances are to be increased by 5 percent.We write

$$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance * 1.05}(account)$$

Now suppose that accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent.We write

$$account \leftarrow \Pi_{AN,BN,\ balance\ *\ 1.06}\ (\sigma_{balance>10000}\ (account)) \cup \Pi_{AN,\ BN\ balance\ *\ 1.05}\ (\sigma_{balance\leq 10000}\ (account))$$

where the abbreviations AN and BN stand for *account-number* and *branch-name*, respectively

3.5 Data Dictionary:

A data dictionary contains metadata i.e data about the database. The data dictionary is very important as it contains information such as what is in the database, who is allowed to access it, where is the database physically stored etc. The users of the database normally don't interact with the data dictionary; it is only handled by the database administrators.

The data dictionary in general contains information about the following −

- Names of all the database tables and their schemas.
- Details about all the tables in the database, such as their owners, their security constraints, when they were created etc.
- Physical information about the tables such as where they are stored and how.
- Table constraints such as primary key attributes, foreign key information etc.
- Information about the database views that are visible.

This is a data dictionary describing a table that contains employee details.

Field Name	Data Type	Field Size for display	Description	Example
Employee Number	Integer	8	Unique ID of each employee	12345678
Name	Text	25	Name of the employee	Hari Khadka
Date of Birth	Date/Time	10	DOB of Employee	08/03/1994
Phone Number	Integer	10	Phone number of employee	1234567891

Chapter 4: Relational Database Query Languages

SQL:

Structured Query Language is a special-purpose programming language designed for managing data held in a RDBMS. Originally based upon relational algebra and relational-calculus constructs, SQL consists of a data definition language and data manipulation language.
The SQL language has several parts:

• **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
• **Interactive data-manipulation language (DML).** The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It includes also commands to insert tuples into, delete tuples from, and modify tuples in the database.
• **View definition.** The SQL DDL includes commands for defining views.
• **Transaction control.** SQL includes commands for specifying the beginning and ending of transactions.
• **Embedded SQL and dynamic SQL.** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.
• **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
• **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

In this chapter, we cover the DML and the basic DDL features of SQL. The enterprise that we use in the examples in this chapter, is a banking enterprise with the following relation schemas:

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

Basic Structure of SQL

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

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

A typical SQL query has the form

select *A*1*, A*2*, . . .,An*
from *r*1*, r*2*, . . . , rm*
where *P*

Each *Ai* represents an attribute, and each *ri* a relation. *P* is a predicate. The query is equivalent to the relational-algebra expression

$\Pi A1, A2,...,An(\sigma P (r1 \times r2 \times \cdots \times rm))$

If the **where** clause is omitted, the predicate *P* is **true**. However, unlike the result of a relational-algebra expression, the result of the SQL query may containmultiple copies of some tuples.

SQL forms the Cartesian product of the relations named in the **from** clause, performs a relational-algebra selection using the **where** clause predicate, and then projects the result onto the attributes of the **select** clause.

The select Clause:

Let us consider a simple query using our banking example, "Find the names of all branches in the *loan* relation":

select *branch-name*
from *loan*

The result is a relation consisting of a single attribute with the heading *branch-name*. It display all branch-name with duplicates also if there any.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as

select distinct *branch-name*
from *loan*

if we want duplicates removed.

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

select all *branch-name*
from *loan*

Since duplicate retention is the default, we will not use **all** in the examples. The asterisk symbol **" * "** can be used to denote "all attributes." A select clause of the form **select** * indicates that all attributes of all relations appearing in the **from** clause are selected.

The **select** clause may also contain arithmetic expressions involving the operators +, −, *, and / operating on constants or attributes of tuples. For example, the query

select *loan-number*, *branch-name*, *amount* * 100
from *loan*

will return a relation that is the same as the *loan* relation, except that the attribute *amount* is multiplied by 100.

The where Clause:

Consider the query "Find all loan numbers for loans made at the Perryridge branch with loan amounts greater that $1200." This query can be written in SQL as:

select *loan-number*
from *loan*
where *branch-name* = 'Perryridge' **and** *amount* > 1200

SQL uses the logical connectives **and**, **or**, and **not**—rather than the mathematical symbols ∧, ∨, and ¬—in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.

SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. If we wish to find the loan number of those loans with loan amounts between $90,000 and $100,000, we can use the **between** comparison to write

select *loan-number*
from *loan*
where *amount* **between** 90000 **and** 100000

Similarly, we can use the **not between** comparison operator.

The from Clause:

The **from** clause by itself defines a Cartesian product of the relations in the clause. Since the natural join is defined in terms of a Cartesian product, a selection, and a projection, it is a relatively simple matter to write an SQL expression for the natural join.
We write the relational-algebra expression

Π*customer-name, loan-number, amount* (*borrower* ⨝ *loan*)

for the query "For all customers who have a loan from the bank, find their names,loan numbers and loan amount." In SQL, this query can be written as

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number*

Notice that SQL uses the notation *relation-name.attribute-name*, as does the relational algebra, to avoid ambiguity in cases where an attribute appears in the schema of more than one relation.

Another e.g

"Find the customer names, loan numbers, and loan amounts for all loans at the Perryridge branch." To write this query, we need to state two constraints in the where clause, connected by the logical connective and:

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number* and
branch-name = 'Perryridge'

The Rename Operation:

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

old-name **as** *new-name*

The **as** clause can appear in both the **select** and **from** clauses.
Consider again the query that we used earlier:

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number*

The result of this query is a relation with the following attributes:

customer-name, loan-number, amount.

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

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

Tuple Variables:

The **as** clause is particularly useful in defining the notion of tuple variables, as is done in the tuple relational calculus. A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the **from** clause by way of the **as** clause. To illustrate, we rewrite the query "For all customers who have a loan from the bank, find their names, loan numbers, and loan amount" as

select *customer-name, T.loan-number, S.amount*
from *borrower* **as** *T*, *loan* **as** *S*
where *T.loan-number = S.loan-number*

Note that we define a tuple variable in the **from** clause by placing it after the name of the relation with which it is associated, with the keyword **as** in between (the keyword **as** is optional).

Tuple variables are most useful for comparing two tuples in the same relation. Suppose that we want the query "Find the names of all branches that have assets greater than at least one branch located in Brooklyn."We can write the SQL expression

select distinct *T.branch-name*
from *branch* **as** *T*, *branch* **as** *S*
where *T.assets > S.assets* **and** *S.branch-city* = 'Brooklyn'

Observe that we could not use the notation *branch.asset*, since it would not be clear which reference to *branch* is intended.

String Operations:

SQL specifies strings by enclosing them in single quotes, for example, 'Perryridge'. The most commonly used operation on strings is pattern matching using the operator **like**. We describe patterns by using two special characters:

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

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

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

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all customers whose street address includes the substring 'Main'." This query can be written as

select *customer-name*
from *customer*
where *customer-street* **like** '%Main%'

Set Operations:

The SQL operations union, intersect, and except operate on relations and correspond to the relational-algebra operations $\cup$, $\cap$, and $-$.

The Union Operation:

To find all customers having a loan, an account, or both at the bank, we write

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

The **union** operation automatically eliminates duplicates, unlike the **select** clause. If we want to retain all duplicates, we must write **union all** in place of **union**:

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

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both *d* and *b*. Thus, if Jones has three accounts and two loans at the bank, then there will be five tuples with the name Jones in the result.

The Intersect Operation:

To find all customers who have both a loan and an account at the bank, we write

(**select distinct** *customer-name*
from *depositor*)
intersect
(**select distinct** *customer-name*
from *borrower*)

The **intersect** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

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

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both *d* and *b*. Thus, if Jones has three accounts and two loans at the bank, then there will be two tuples with the name Jones in the result.

The Except Operation:

To find all customers who have an account but no loan at the bank, we write

(**select distinct** *customer-name*
from *depositor*)
except
(**select** *customer-name*
from *borrower*)

The **except** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **except all** in place of **except**:

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

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies of the tuple in *d* minus the number of duplicate copies of the tuple in *b*, provided that the difference is positive. Thus, if Jones has three accounts and one loan at the bank, then there will be two tuples with the name Jones in the result. If, instead, this customer has two accounts and three loans at the bank, there will be no tuple with the name Jones in the result.

Aggregate Functions:

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

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well. As an illustration, consider the query "Find the average account balance at the Perryridge branch."We write this query as follows:

select avg (*balance*)
from *account*
where *branch-name* = 'Perryridge'

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average account balance at each branch."We write this query as follows:

select *branch-name*, **avg** (*balance*)
from *account*
group by *branch-name*

Retaining duplicates is important in computing an average. There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query "Find the number of depositors for each branch." In this case, a depositor counts only once, regardless of the number of accounts that depositor may have .We write this query as follows:

select *branch-name*, **count** (**distinct** *customer-name*)
from *depositor, account*
where *depositor.account-number* = *account.account-number*
group by *branch-name*

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those branches where the average account balance is more than $1200. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL.

select *branch-name*, **avg** (*balance*)
from *account*
group by *branch-name*
having avg (*balance*) > 1200

If a **where** clause and a **having** clause appear in the same query, SQL applies the predicate in the **where** clause first. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause. SQL then applies the **having** clause, if it is present, to each group; it removes the groups that do not satisfy the **having** clause predicate. The **select** clause uses the remaining groups to generate tuples of the result of the query.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "Find the average balance for each customer who lives in Harrison and has at least three accounts."

select *depositor.customer-name*, **avg** (*balance*)
from *depositor, account, customer*
where *depositor.account-number* = *account.account-number* **and**
depositor.customer-name = *customer.customer-name* **and**
customer-city = 'Harrison'
group by *depositor.customer-name*
having count (**distinct** *depositor.account-number*) >= 3

Nested Subqueries:

SQL provides a mechanism for nesting subqueries. A subquery is a select-from-where expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality.

Set Membership: SQL draws on the relational calculus for operations that allow testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a select clause. The **not in** connective tests for the absence of set membership.
As an illustration, reconsider the query "Find all customers who have both a loan and an account at the bank." We begin by finding all account holders, and we write the subquery

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

We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the subquery. We do so by nesting the subquery in an outer **select**. The resulting query is now

select distinct *customer-name*
from *borrower*
where *customer-name* **in** (**select** *customer-name*
from *depositor*)

In the preceding example, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. We can thus write the query "Find all customers who have both an account and a loan at the Perryridge branch" in yet another way:

select distinct *customer-name*
from *borrower, loan*
where *borrower.loan-number* = *loan.loan-number* **and**
branch-name = 'Perryridge' **and**
(*branch-name, customer-name*) **in**
(**select** *branch-name, customer-name*
from *depositor, account*
where *depositor.account-number* = *account.account-number*)

We use the **not in** construct in a similar way. For example, to find all customers who do have a loan at the bank, but do not have an account at the bank, we can write

select distinct *customer-name*
from *borrower*
where *customer-name* **not in** (**select** *customer-name*
from *depositor*)

Set Comparison: As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all branches that have assets greater than those of at least one branch located in Brooklyn." Previously(**in tuple variables example**) we wrote this query as follows:

select distinct *T.branch-name*
from *branch* as *T*, *branch* **as** *S*
where *T.assets* > *S.assets* **and** *S.branch-city* = 'Brooklyn'

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > **some**. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

select *branch-name*
from *branch*
where *assets* > **some** (**select** *assets* **from** *branch* **where** *branch-city* = 'Brooklyn')

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and<> **some** comparisons.
Now we modify our query slightly. Let us find the names of all branches that have an asset value greater than that of each branch in Brooklyn. The construct > **all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

```
select branch-name
from branch
where assets > all (select assets
                    from branch
                    where branch-city = 'Brooklyn')
```

As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons

Test for Empty Relations:

SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty. Using the **exists** construct, we can write the query "Find all customers who have both an account and a loan at the bank" in still another way:

```
select customer-name
from borrower
where exists (select * from depositor
              where depositor.customer-name = borrower.customer-name)
```

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct. To illustrate the **not exists** operator, consider again the query "Find all customers who have an account at all the branches located in Brooklyn." For each customer, we need to see whether the set of all branches at which that customer has an account contains the set of all branches in Brooklyn. Using the **except** construct, we can write the query as follows:

```
select distinct S.customer-name
from depositor as S
where not exists ((select branch-name  from branch
                   where branch-city = 'Brooklyn')
                  except
                  (select R.branch-name
                  from depositor as T, account as R
                  where T.account-number = R.account-number and S.customer-name =
                        T.customer- name))
```

Views:

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is

```
create view v as <query expression>
```

where <query expression> is any legal query expression. The view name is represented by *v.*
As an example, consider the view consisting of branch names and the names of customers who have either an account or a loan at that branch. Assume that we want this view to be called *all-customer*.We define this view as follows:

```
create view all-customer as
        (select branch-name, customer-name
        from depositor, account
        where depositor.account-number = account.account-number)
    union
        (select branch-name, customer-name
        from borrower, loan
```

where *borrower.loan-number* = *loan.loan-number*)

Using the view *all-customer*, we can find all customers of the Perryridge branch by writing

select *customer-name*
from *all-customer*
where *branch-name* = 'Perryridge'

Modification of the Database:

Now, we discuss how to add, remove, or change information with SQL.

Deletion

A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

delete from *r*
where *P*

where *P* represents a predicate and *r* represents a relation. A **delete** command operates on only one relation.

Here are examples of SQL delete requests:

- Delete all account tuples in the Perryridge branch.

  delete from *account*
  where *branch-name* = 'Perryridge'

- Delete all loans with loan amounts between $1300 and $1500.

  delete from *loan*
  where *amount* **between** 1300 **and** 1500

- Delete all account tuples at every branch located in Needham.

  delete from *account*
  where *branch-name* **in** (**select** *branch-name*
  from *branch*
  where *branch-city* = 'Needham')

This **delete** request first finds all branches in Needham, and then deletes all *account* tuples pertaining to those branches.

Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** nested in the **where** clause of a **delete**. The **delete** request can contain a nested **select** that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all accounts with balances below the average at the bank.We could write

delete from *account*
where *balance* < (**select avg** (*balance*)
from *account*)

Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The attribute values for inserted tuples must be members of the attribute's domain.

The simplest **insert** statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is an account A-9732 at the Perryridge branch and that is has a balance of $1200.We write

insert into *account*
values ('A-9732', 'Perryridge', 1200)

or we can also write as

insert into *account* (*account-number, branch-name, balance*)

values ('A-9732', 'Perryridge', 1200) **OR**
insert into *account* (*branch-name, account-number, balance*)
values ('Perryridge', 'A-9732', 1200)

Updates

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used.
Suppose that annual interest payments are being made, and all balances are to be increased by 5 percent.We write

update *account*
set *balance* = *balance* * 1.05

If interest is to be paid only to accounts with a balance of $1000 or more, we can write

update *account*
set *balance* = *balance* * 1.05
where *balance* >= 1000

Another example: "Pay 5 percent interest on accounts whose balance is greater than average"

update *account*
set *balance* = *balance* * 1.05
where *balance* > **select avg** (*balance*)
from *account*

Joined Relations:

SQL provides not only the basic Cartesian-product mechanism for joining tuples of relations , but, SQL also provides various other mechanisms for joining relations, including condition joins and natural joins, as well as various forms of outer joins. These additional operations are typically used as subquery expressions in the **from** clause.

loan

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Figure 4.1 The *loan* and *borrower* relations.

Examples:

We illustrate the various join operations by using the relations *loan* and *borrower* in Figure 4.1. We start with a simple example of inner joins. Figure 4.2 shows the result of the expression

loan **inner join** *borrower* **on** *loan.loan-number* = *borrower .loan-number*

The expression computes the theta join of the *loan* and the *borrower* relations, with the join condition being *loan.loan-number* = *borrower.loan-number*. The attributes of the result consist of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation. Note that the attribute *loan-number* appears twice in the figure—the first occurrence is from *loan*, and the second is from *borrower*. The SQL standard does not require attribute names in such results to be unique. An **as** clause should be used to assign unique names to attributes in query and subquery results. We rename the result relation of a join and the attributes of the result relation by using an **as** clause, as illustrated here:

loan **inner join** *borrower* **on** *loan.loan-number* = *borrower.loan-number*
as *lb(loan-number, branch, amount, cust, cust-loan-num)*

We rename the second occurrence of *loan-number* to *cust-loan-num*. The ordering of the attributes in the result of the join is important for the renaming.

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

Figure 4.2 The result of *loan* **inner join** *borrower* **on** *loan.loan-number = borrower.loan-number*.

Next, we consider an example of the **left outer join** operation:

loan **left outer join** *borrower* **on** *loan.loan-number* = *borrower.loan-number*

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Figure 4.3 The result of *loan* **left outer join** *borrower* **on** *loan.loan-number = borrower.loan-number*.

Now, we consider an example of the **natural join** operation:

loan **natural inner join** *borrower*

This expression computes the natural join of the two relations. The only attribute name common to *loan* and *borrower* is *loan-number*. Figure 4.4 shows the result of the expression.

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Figure 4.4 The result of *loan* **natural inner join** *borrower*.

Each of the variants of the join operations in SQL consists of a *join type* and a *join condition*. The join condition defines which tuples in the two relations match and what attributes are present in the result of the join. The join type defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated. Figure 4.5 shows some of the allowed join types and join conditions.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using ($A_1, A_1, \ldots, A_n$)

Figure 4.5 Join types and join conditions.

The meaning of the join condition **natural**, in terms of which tuples from the two relations match, is straightforward. The ordering of the attributes in the result of a natural join is as follows. The join attributes (that is, the attributes common to both relations) appear first, in the order in which they appear in the left-hand-side relation. Next come all nonjoin attributes of the left-hand-side relation, and finally all nonjoin attributes of the right-hand-side relation.

The **right outer join** is symmetric to the **left outer join**. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join.

Here is an example of combining the natural join condition with the right outer join type:

loan **natural right outer join** *borrower*

Figure 4.6 shows the result of this expression.

The **full outer join** is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls tuples from the left-hand-side relation that did not match with any from the right-hand-side, and adds them to the result. Similarly, it extends with nulls tuples from the right-handside relation that did not match with any tuples from the left-hand-side relation and adds them to the result.

For example, Figure 4.7 shows the result of the expression

loan **full outer join** *borrower* **using** (loan-number)

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Figure 4.6 The result of *loan* **natural right outer join** *borrower*.

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Figure 4.7 The result of *loan* **full outer join** *borrower* **using**(*loan-number*).

QBE(Query-by-Example)

QBE is a graphical language, where queries look like tables. **Query-by-Example (QBE)** is the name of both a data-manipulation language and an early database system that included this language. Here, we consider only the data-manipulation language. It has two distinctive features:

1. Unlike most query languages and programming languages, QBE has a **two-dimensional syntax**: Queries *look* like tables. A query in a one-dimensional language (for example, SQL) *can* be written in one (possibly long) line. A twodimensional language *requires* two dimensions for its expression.

2. QBE queries are expressed "by example." Instead of giving a procedure for obtaining the desired answer, the user gives an example of what is desired.

We express queries in QBE by **skeleton tables**. These tables show the relation schema, as in Figure 5.1. Rather than clutter the display with all skeletons, the user selects those skeletons needed for a given query and fills in the skeletons with **example rows**. An example row consists of constants and *example elements*, which are domain variables. To avoid confusion between the two, QBE uses an underscore character (_) before domain variables, as in _*x*, and lets constants appear without any qualification.

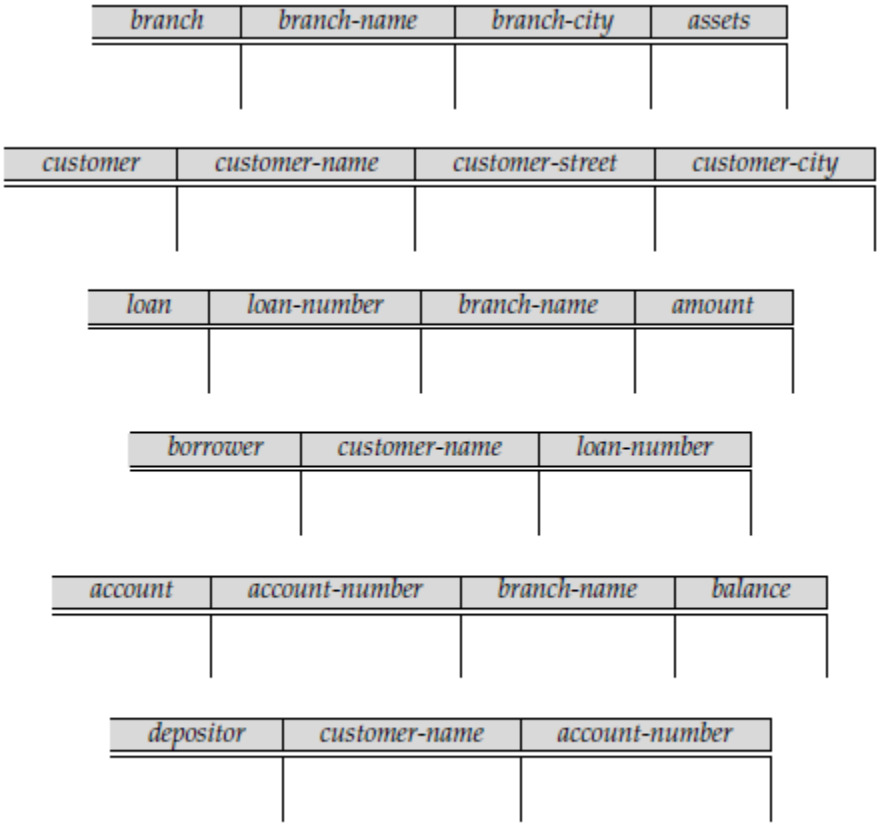


Figure 5.1 QBE skeleton tables for the bank example.

Queries on One Relation

To find all loan numbers at the Perryridge branch, we bring up the skeleton for the *loan* relation, and fill it in as follows:

loan	*loan-number*	*branch-name*	*amount*
	P._*x*	Perryridge	

This query tells the system to look for tuples in *loan* that have "Perryridge" as the value for the *branch-name* attribute. For each such tuple, the system assigns the value of the *loan-number* attribute to the variable *x*.

QBE assumes that a blank position in a row contains a unique variable. As a result, if a variable does not appear more than once in a query, it may be omitted. Our previous query could thus be rewritten as

loan	*loan-number*	*branch-name*	*amount*
	P.	Perryridge	

QBE (unlike SQL) performs duplicate elimination automatically. To suppress duplicate elimination, we insert the command ALL. after the P. command:

loan	*loan-number*	*branch-name*	*amount*
	P.ALL.	Perryridge	

To display the entire *loan* relation,

loan	*loan-number*	*branch-name*	*amount*
P.			

QBE allows queries that involve arithmetic comparisons (for example, >), "Find the loan numbers of all loans with a loan amount of more than $700":

loan	*loan-number*	*branch-name*	*amount*
	P.		>700

The arithmetic operations that QBE supports are =, <, ≤, >, ≥, and ¬.for example "Find the names of all branches that are not located in Brooklyn." This query can be written as follows:

branch	*branch-name*	*branch-city*	*assets*
	P.	¬ Brooklyn	

Consider the query "Find the loan numbers of all loansmade jointly to Smith and Jones":

borrower	*customer-name*	*loan-number*
	"Smith"	P._x
	"Jones"	_x

Queries on Several Relations

QBE allows queries that span several different relations (analogous to Cartesian product or natural join in the relational algebra). The connections among the various relations are achieved through variables that force certain tuples to have the same value on certain attributes. As an illustration, suppose that we want to find the names of all customers who have a loan from the Perryridge branch. This query can be written as

loan	*loan-number*	*branch-name*	*amount*
	_x	Perryridge	

borrower	*customer-name*	*loan-number*
	P._y	_x

To evaluate the preceding query, the system finds tuples in *loan* with "Perryridge" as the value for the *branch-name* attribute. For each such tuple, the system finds tuples in *borrower* with the same value for the *loan-number* attribute as the *loan* tuple. It displays the values for the *customer-name* attribute.

Chapter 5: Database Constraints and Relational Database Design

INTEGRITY CONSTRAINT An integrity constraint is a mechanism used by dbms to prevent invalid data entry into the table. It has enforcing the rules for the columns in a table. The types of the integrity constraints are: a) Domain Integrity b) Entity Integrity c) Referential Integrity.

a) Domain Integrity This constraint sets a range and any violations that take place will prevent the user from performing the manipulation that caused the breach. It includes:

Not Null constraint: While creating tables, by default the rows can have null value .the enforcement of not null constraint in a table ensure that the table contains values.

Principle of null values:

- Setting null value is appropriate when the actual value is unknown, or when a value would not be meaningful.
- A null value is not equivalent to a value of zero.
- A null value will always evaluate to null in any expression.
- When a column name is defined as not null, that column becomes a mandatory i.e., the user has to enter data into it.
- Not null Integrity constraint cannot be defined using the alter table command when the table contain rows.

Check Constraint: Check constraint can be defined to allow only a particular range of values .when the manipulation violates this constraint, the record will be rejected. Check condition cannot contain sub queries.

E.g: Create table student (regno int, mark int, constraint b check (mark >=0 and mark <=100));

b) Entity Integrity Maintains uniqueness in a record. An entity represents a table and each row of a table represents an instance of that entity. To identify each row in a table uniquely we need to use this constraint. There are 2 entity constraints:

Unique key constraint It is used to ensure that information in the column for each record is unique, as with telephone or drivers license numbers. It prevents the duplication of value with rows of a specified column in a set of column. A column defined with the constraint can allow null value. If unique key constraint is defined in more than one column i.e., combination of column cannot be specified.Maximum combination of columns that a composite unique key can contain is 16.

Primary Key Constraint A primary key avoids duplication of rows and does not allow null values. It can be defined on one or more columns in a table and is used to uniquely identify each row in a table. These values should never be changed and should never be null. A table should have only one primary key. If a primary key constraint is assigned to more than one column or combination of column is said to be composite primary key, which can contain 16 columns.

c) Referential Integrity It enforces relationship between tables. To establish parent-child relationship between 2 tables having a common column definition, we make use of this

constraint. To implement this, we should define the column in the parent table as primary key and same column in the child table as foreign key referring to the corresponding parent entry.

Foreign key A column or combination of column included in the definition of referential integrity, which would refer to a referenced key.**Referenced key** It is a unique or primary key upon which is defined on a column belonging to the parent table.

e.g :

```
CREATE TABLE department
(D_id int,
 D_name varchar(30),
 CONSTRAINT pk_nid Primary key(D_id));

CREATE TABLE teacher
(
T_id int  PRIMARY KEY,
D_id int,
T_name varchar(50) NOT NULL,
T_address varchar(50),
T_email varchar(25),
CONSTRAINT fk_td FOREIGN KEY (D_id) REFERENCES department(D_id))
```

Cascading Actions in SQL

create table *teacher*
. . .
foreign key*(D_id)* **references** *department(D_id))*
on delete cascade
on update cascade
. . . **)**

- Due to the **on delete cascade** clauses, if a delete of a tuple in *branch* results in referential-integrity constraint violation, the delete "cascades" to the *account* relation, deleting the tuple that refers to the branch that was deleted.
- Cascading updates are similar.
- If there is a chain of foreign-key dependencies across multiple relations, with **on delete cascade** specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain.
- If a cascading update to delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.
- Referential integrity is only checked at the end of a transaction
    - ✓ Intermediate steps are allowed to violate referential integrity provided later steps remove the violation

- ✓ Otherwise it would be impossible to create some database states, e.g. insert two tuples whose foreign keys point to each other (e.g. *spouse* attribute of relation *marriedperson)*

- Alternative to cascading:
  - **on delete set null**
- Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null**
  - ✓ if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint!

Assertion:

. An *assertion* is a predicate expressing a condition that we wish the database always to satisfy.

- ✓ An assertion in SQL takes the form

create assertion <assertion-name> **check** <predicate>

. When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion

- ✓ This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

.Asserting for all X, P(X) is achieved in a round-about fashion using not exists X such that not P(X)

Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum-constraint check
  (not exists (select * from branch
                  where (select sum(amount) from loan
                              loan.branch-name = branch.branch-name)
                    >= (select sum(amount) from account
                              where loan.branch-name = branch.branch-name)))
```

Triggers

A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:

- Specify the conditions under which the trigger is to be executed.
- Specify the actions to be taken when the trigger executes.

Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

Statement Level Triggers

Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a single transaction

- Use **for each statement** instead of **for each row**
- Use **referencing old table** or **referencing new table** to refer to temporary tables containing the affected rows
- Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

Trigger Example:

For creating a new trigger, we need to use the CREATE TRIGGER statement. Its syntax is as follows −

```
CREATE TRIGGER trigger_name trigger_time trigger_event

ON table_name

FOR EACH ROW

BEGIN

...

END;
```

Here,

- **Trigger_name** is the name of the trigger which must be put after the CREATE TRIGGER statement. The naming convention for trigger_name can be like [trigger time]_[table name]_[trigger event]. For example, before_student_update or after_student_insert can be the name of the trigger.
- **Trigger_time** is the time of trigger activation and it can be BEFORE or AFTER. We must have to specify the activation time while defining a trigger. We must use BEFORE if we want to process action prior to the change made on the table and AFTER if we want to process action post to the change made on the table.
- **Trigger_event** can be INSERT, UPDATE, or DELETE. This event causes the trigger to be invoked. A trigger only can be invoked by one event. To define a trigger that is invoked by multiple events, we have to define multiple triggers, one for each event.
- **Table_name** is the name of the table. Actually, a trigger is always associated with a specific table. Without a table, a trigger would not exist hence we have to specify the table name after the 'ON' keyword.
- **BEGIN…END** is the block in which we will define the logic for the trigger.
  e.g: Suppose we want to apply trigger on the table Student_age which is created as follows –

```
Create table Student_age(age INT, Name Varchar(35));
```

Now, the following trigger will automatically insert the age = 0 if someone tries to insert age < 0.

```
DELIMITER //

Create Trigger before_inser_studentage BEFORE INSERT ON student_age
FOR EACH ROW

BEGIN

IF NEW.age < 0 THEN SET NEW.age = 0;

END IF;

END //
```

Now, for invoking this trigger, we can use the following statements –

```
INSERT INTO Student_age(age, Name) values(30, 'Rahul');

INSERT INTO Student_age(age, Name) values(-10, 'Harshit');

Select * from Student_age;
```

```
+------+---------+
| age  | Name    |
+------+---------+
| 30   | Rahul   |
| 0    | Harshit |
+------+---------+
```

The above result set shows that on inserting the negative value in the table will lead to insert 0 by a trigger.

CLOSURE OF A SET OF FUNCTIONAL DEPEDENCIES

Given a relational schema R, a functional dependencies f on R is logically implied by a set of functional dependencies F on R if every relation instance r(R) that satisfies F also satisfies f.

The closure of F, denoted by F^+, is the set of all functional dependencies logically implied by F.

The closure of F can be found by using a collection of rules called **Armstrong axioms.**

- **Reflexivity rule:** If A *is a set of attributes and B is subset or equal to A, then A*→B holds.
- **Augmentation rule:** If A→B holds and C is a set of attributes, then CA→CB holds
- **Transitivity rule:** If A→B holds and B→C holds, then A→C holds.
- **Union rule:** If A→B holds and A→C then A→BC holds
- **Decomposition rule:** If A→BC holds, then A→B holds and A→C holds.
- **Pseudo transitivity rule:** If A→B holds and BC→D holds, then AC→D holds.

Suppose we are given a relation schema R=(A,B,C,G,H,I) and the set of function dependencies

A→B,A→C,CG→H,CG→I,B→H

- We list several members of F^+ here:
  - ☐ A→H, since A→B and B→H hold, we apply the transitivity rule.
  - ☐ CG→HI. Since CG→H and CG→I , the union rule implies that CG→HI
  - ☐ AG→I, since A→C and CG→I, the pseudo transitivity rule implies that AG→I holds

. A→BC, since A→B and A→C hold, we apply union rule

. AG→H, since A→C and CG→H, the pseudo transitivity rule implies that AG→H holds

Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has *n* attributes *A*1, *A*2, ...,*An*; let us think of the whole database as being described by a single **universal** relation schema *R* = {*A*1, *A*2, ... , *An*}. We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.

Definition. *A **functional dependency**, denoted by X → 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**.

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.

Normalization:

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

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. 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. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

Consider the DEPARTMENT relation schema shown in Figure below, whose primary key is Dnumber. We assume that each department can have *a number of* locations. As we can see, this is not in 1NF because Dlocations is not an atomic attribute.

DEPARTMENT

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

To Convert it into 1NF:

There are two 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 below. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

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

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 15.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation.

DEPARTMENT

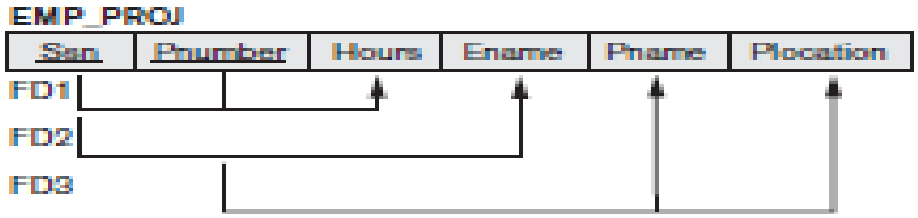
Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Second Normal Form:

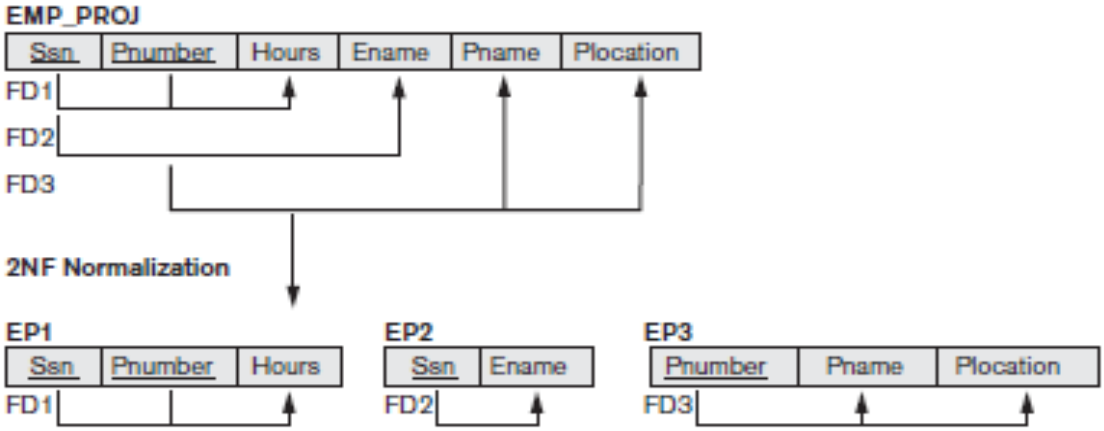
Second normal form (2NF) is based on the concept of *full functional dependency.* A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \varepsilon X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \varepsilon X$ can be removed from X and the dependency still holds; that is, for some $A \varepsilon X$, $(X - \{A\}) \rightarrow Y$. In Figure EMP_PROJ, {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.*

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure below is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.



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 of above Figure lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure below each of which is in 2NF.

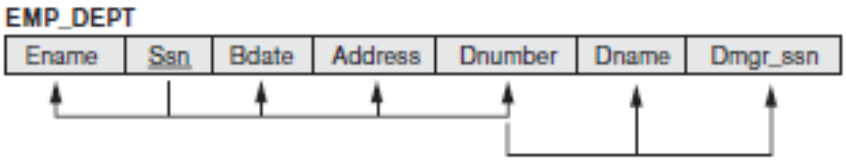


Third Normal Form:

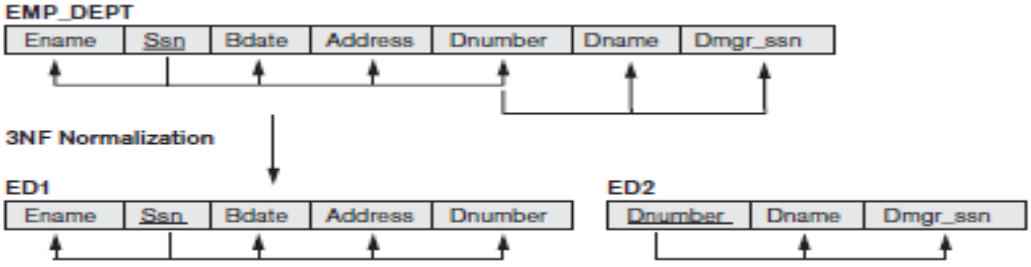
Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ 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 below 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 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.



Summary of Normal Forms Based on Primary Key and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

Boyce-Codd Normal Form (BCNF):

- A relation is in BCNF if every determinant is a candidate key
- Those determinants that are keys we initially call *candidate keys*.
- Eventually, we select a single candidate key to be *the key* for the relation.
- **Consider the following example:**
  - **Funds consist of one or more Investment Types.**
  - **Funds are managed by one or more Managers**
  - **Investment Types can have one more Managers**
  - **Managers only manage one type of investment.**
- **Relation: FUNDS (FundID, InvestmentType, Manager)**

Fund ID	Investment Type	Manager
99	Common Stock	Smith
33	Common Stock	Green
22	Growth Stocks	Brown

11	**Municipal Bonds**	**Smith**

- **FD1: FundID, InvestmentType → Manager**
- **FD2: FundID, Manager → InvestmentType**
- **FD3: Manager → InvestmentType**
- **In this case, the combination FundID and InvestmentType form a *candidate key* because we can use FundID,InvestmentType to uniquely identify a tuple in the relation.**
- **Similarly, the combination FundID and Manager also form a *candidate key* because we can use FundID, Manager to uniquely identify a tuple.**
- **Manager by itself is not a candidate key because we cannot use Manager alone to uniquely identify a tuple in the relation.**
- **Is this relation FUNDS(FundID, InvestmentType, Manager) in 1NF, 2NF or 3NF ?**

Given we pick FundID, InvestmentType as the *Primary Key:* 1NF for sure. 2NF because all of the non-key attributes (Manager) is dependant on all of the key. 3NF because there are no transitive dependencies.

- **Therefore, while FUNDS relation is in 1NF, 2NF and 3NF, it is in BCNF because not all determinants (Manager in FD3) are candidate keys.**
- **The following are steps to normalize a relation into BCNF:**
    1. **List all of the determinants.**
    2. **See if each determinant can act as a key (candidate keys).**
    3. **For any determinant that is *not* a candidate key, create a new relation from the functional dependency. Retain the determinant in the original relation.**
- **For our example:**

FUNDS (FundID, InvestmentType, Manager)

1. The determinants are:

FundID, InvestmentType
FundID, Manager
Manager

2. Which determinants can act as keys ?

FundID, InvestmentType *YES*
FundID, Manager *YES*
Manager *NO*

3. Create a new relation from the functional dependency:

MANAGERS(Manager, InvestmentType)
FUND_MANAGERS(FundID, Manager)

In this last step, we have retained the determinant "Manager" in the original relation MANAGERS.

- **Each of the new relations sould be checked to ensure they meet the definitions of 1NF, 2NF, 3NF and BCNF**

Multivalued Dependency and Fourth Normal Form:

If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP shown in Figure 1(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 spec fied as a multivalued dependency on the EMP relation, which we define in this section. 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

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

Figure 1:Multivalued dependency and fourth normal form.(a) The Emp relation with two MVDs:Ename→→PNAME and ENAME→→DNAME. (b) Decomposing EMP into two relations in 4NF

Formal Definition of Multivalued Dependency:

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 *t*1 and *t*2 exist in *r* such that *t*1[X] = *t*2[X], then two tuples *t*3 and *t*4 should also exist in *r* with the following properties, where we use *Z* to denote (*R* – (*X* ∪Y)):

- *t*3[*X*] = *t*4[*X*] = *t*1[*X*] = *t*2[*X*].
- *t*3[*Y*] = *t*1[*Y*] and *t*4[*Y*] = *t*2[*Y*].
- *t*3[*Z*] = *t*2[*Z*] and *t*4[*Z*] = *t*1[*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 MVD** if (a) *Y* is a subset of *X*, or (b) *X* ∪Y = *R*. For example, the relation EMP_PROJECTS in Figure 1(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. In the EMP relation of Figure 1(a), the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname).

Fourth Normal Form:

Definition. A relation schema *R* is in 4**NF** 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*.

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 1(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 1(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.

To illustrate the importance of 4NF, Figure 16.4(a) shows the EMP relation in Figure 1(a) with an additional employee, 'Brown', who has three dependents ('Jim', 'Joan', and 'Bob') and works on four different projects ('W', 'X', 'Y', and 'Z'). There are 16 tuples in EMP in Figure 16.4(a). If we decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, as shown in Figure 16.4(b), we need to store a total of only 11 tuples in both relations. Not only would the decomposition save on storage, but the update anomalies associated with multivalued dependencies would also be avoided. For example, if 'Brown' starts working on a new additional project 'P,'we must insert *three* tuples in EMP—one for each dependent. If we forget to insert any one of those, the relation violates the MVD and becomes inconsistent in that it incorrectly implies a relationship between project and dependent. If the relation has nontrivial MVDs, then insert, delete, and update operations on single tuples may cause additional tuples to be modified besides the one in question. If the update is handled incorrectly, the meaning of the relation may change. However, after normalization into 4NF, these update anomalies disappear. For example, to add the information that 'Brown' will be assigned to project 'P', only a single tuple need be inserted in the 4NF relation EMP_PROJECTS.

Figure 16.4
Decomposing a relation state of EMP that is not in 4NF. (a) EMP relation with additional tuples. (b) Two corresponding 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

(a) **EMP**

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) **EMP_PROJECTS**

Ename	Pname
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

Ename	Dname
Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

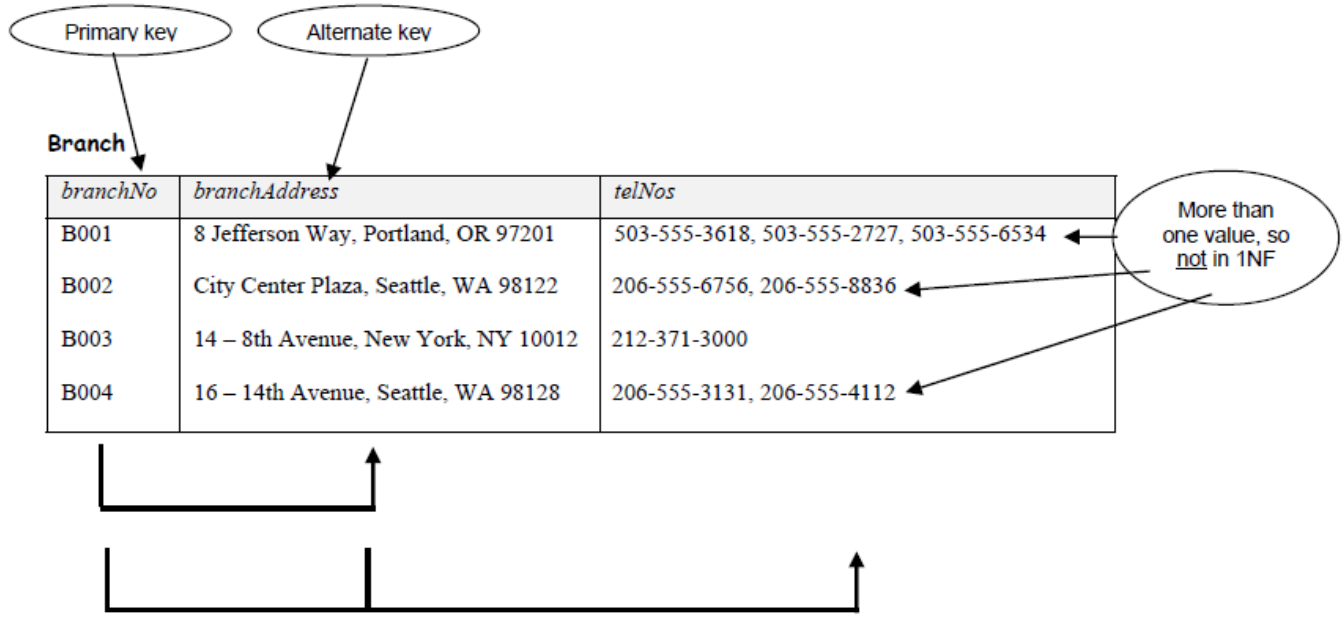
Some Solved Example of Normalization:

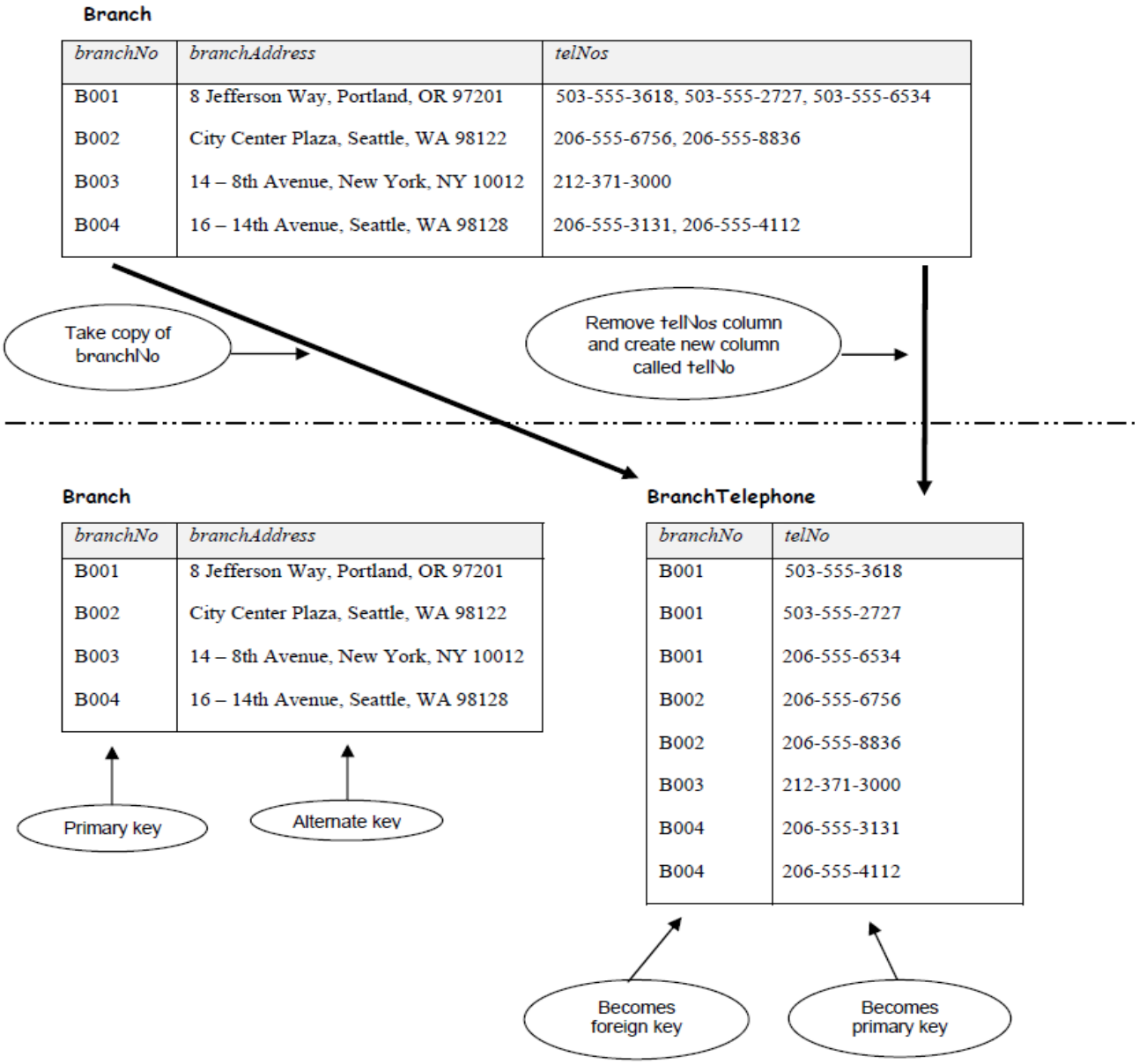
Q1. Examine the table shown below.

branchNo	*branchAddress*	*telNos*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618, 503-555-2727, 503-555-6534
B002	City Center Plaza, Seattle, WA 98122	206-555-6756, 206-555-8836
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131, 206-555-4112

(a) Why is this table not in 1NF?
(b) Describe and illustrate the process of normalizing the data shown in this table to third normal form (3NF).
(c) Identify the primary, alternate and foreign keys in your 3NF relations.

Answer:





Q. 2 Examine the table shown below.

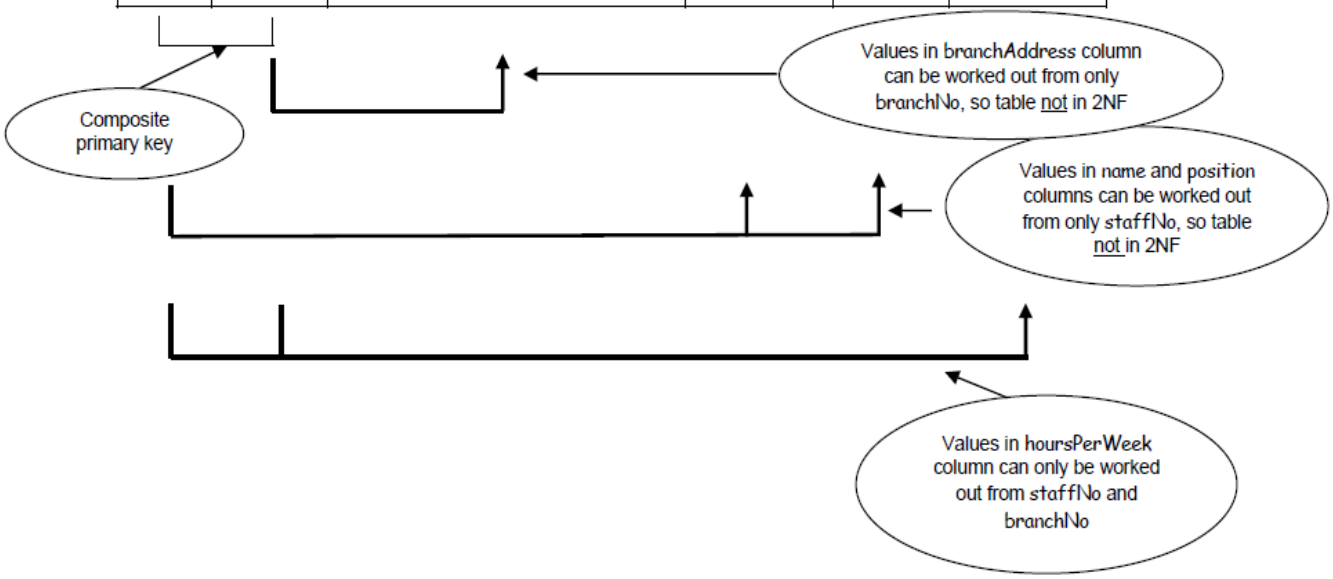
staffNo	*branchNo*	*branchAddress*	*name*	*position*	*hoursPerWeek*
S4555	B002	City Center Plaza, Seattle, WA 98122	Ellen Layman	Assistant	16
S4555	B004	16 – 14th Avenue, Seattle, WA 98128	Ellen Layman	Assistant	9
S4612	B002	City Center Plaza, Seattle, WA 98122	Dave Sinclair	Assistant	14
S4612	B004	16 – 14th Avenue, Seattle, WA 98128	Dave Sinclair	Assistant	10

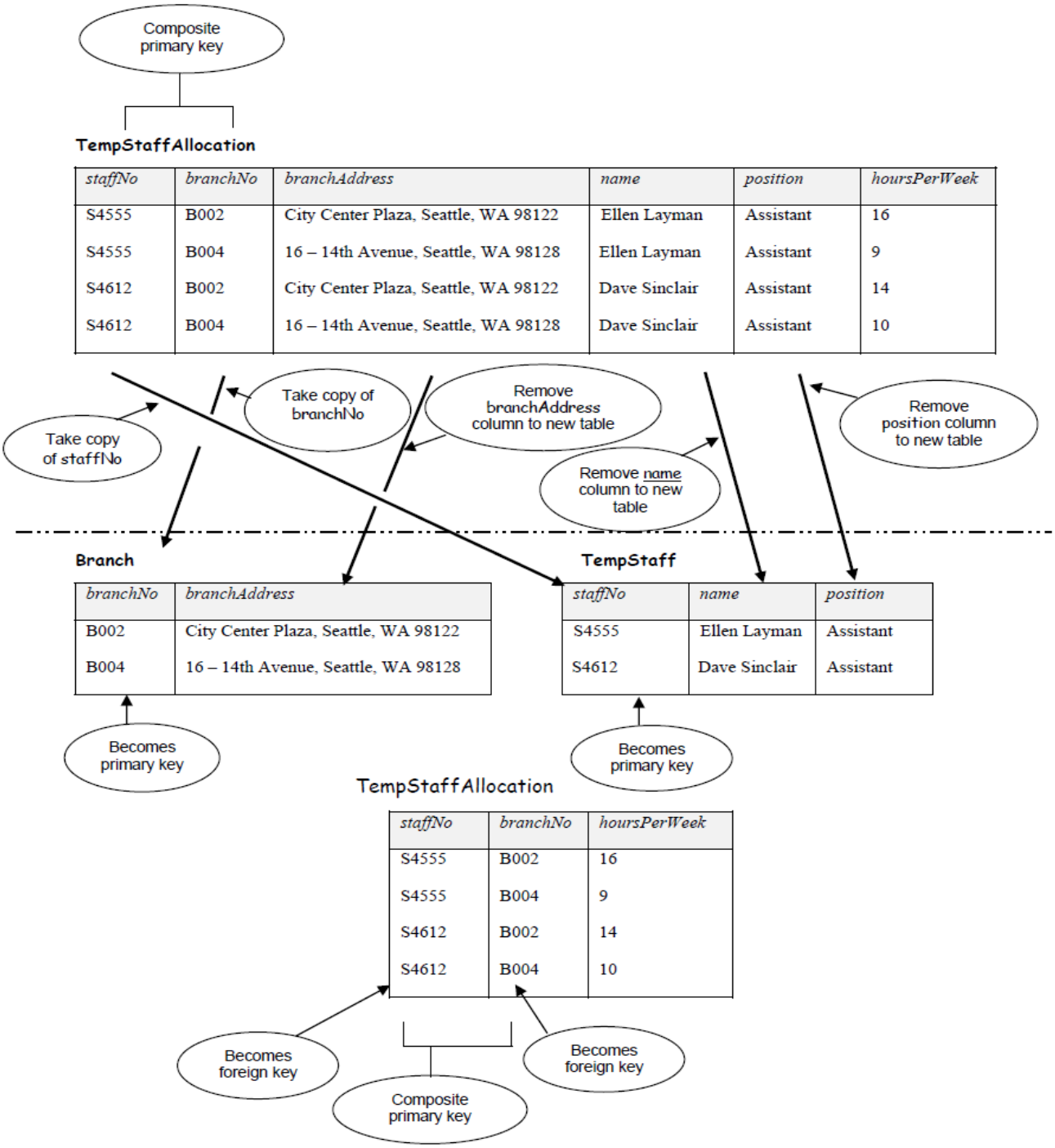
(a) Why is this table not in 2NF?
(b) Describe and illustrate the process of normalizing the data shown in this table to third normal form (3NF).
(c) Identify the primary, (alternate) and foreign keys in your 3NF relations.

Answer:

TempStaffAllocation

staffNo	*branchNo*	*branchAddress*	*name*	*position*	*hoursPerWeek*
S4555	B002	City Center Plaza, Seattle, WA 98122	Ellen Layman	Assistant	16
S4555	B004	16 – 14th Avenue, Seattle, WA 98128	Ellen Layman	Assistant	9
S4612	B002	City Center Plaza, Seattle, WA 98122	Dave Sinclair	Assistant	14
S4612	B004	16 – 14th Avenue, Seattle, WA 98128	Dave Sinclair	Assistant	10





Q.3 Examine the table shown below.

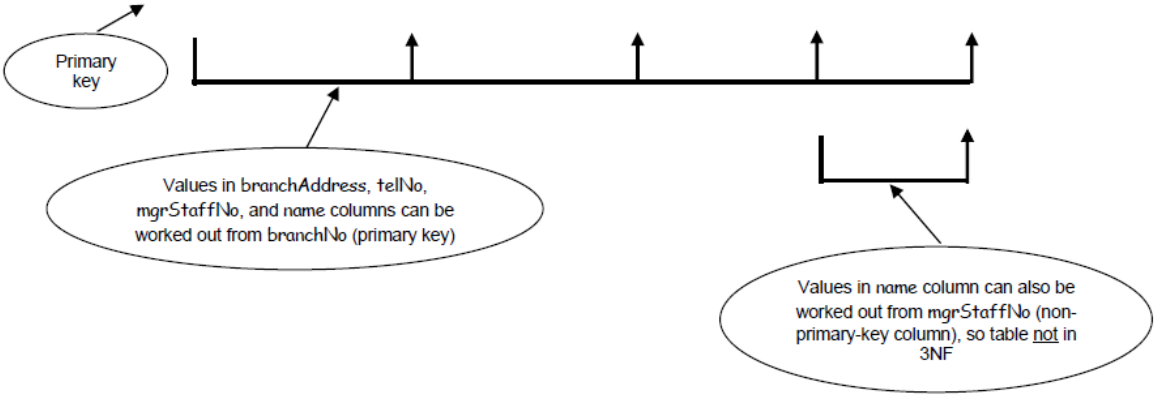
branchNo	*branchAddress*	*telNo*	*mgrStaffNo*	*name*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618	S1500	Tom Daniels
B002	City Center Plaza, Seattle, WA 98122	206-555-6756	S0010	Mary Martinez
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000	S0145	Art Peters
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131	S2250	Sally Stern

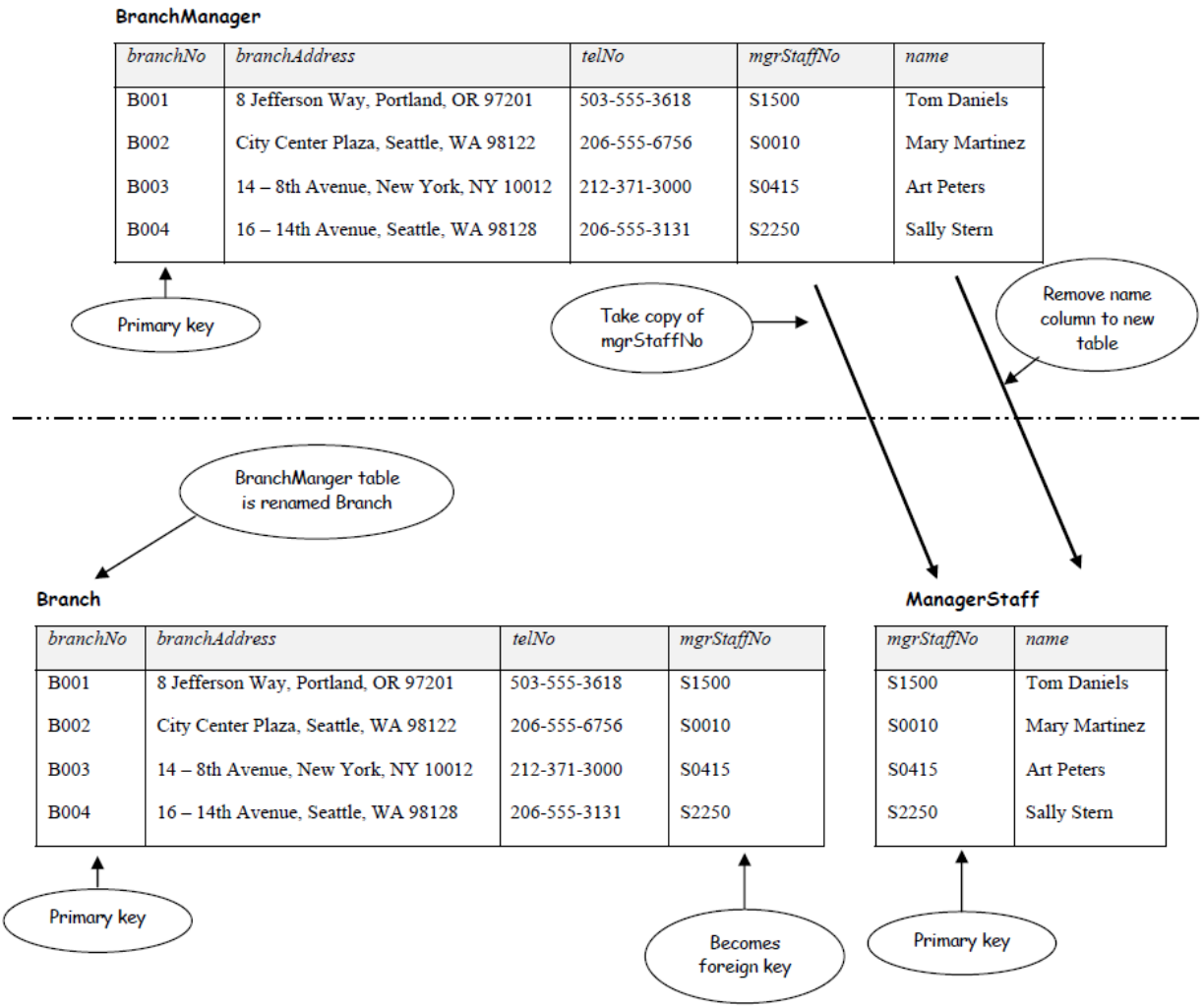
(a) Why is this table not in 3NF?
(b) Describe and illustrate the process of normalizing the data shown in this table to third normal form (3NF).
(c) Identify the primary, (alternate) and foreign keys in your 3NF relations.

Answer:

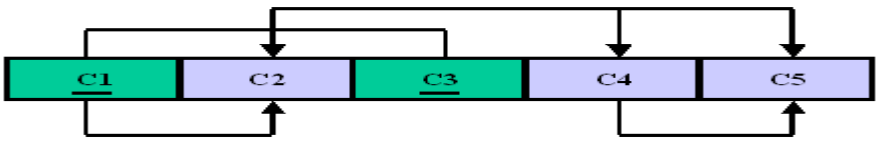
BranchManager

branchNo	*branchAddress*	*telNo*	*mgrStaffNo*	*name*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618	S1500	Tom Daniels
B002	City Center Plaza, Seattle, WA 98122	206-555-6756	S0010	Mary Martinez
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000	S0145	Art Peters
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131	S2250	Sally Stern





Q4. Given the dependency diagram shown in the following Figure, answer the following questions.



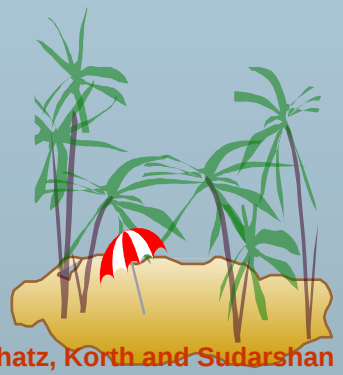
a. Identify and discuss each of the indicated dependencies.

Answer:

- C1 → C2 represents a partial dependency, because C2 depends only on C1, rather than on the entire primary key composed of C1 and C3.
- C4 → C5 represents a transitive dependency, because C5 depends on an attribute (C4) that is not part of a primary key.
- C1, C3 → C2, C4, C5 represents a functional dependency, because C2, C4, and C5 depend on the primary key composed of C1 and C3.

Chapter 6: Security

- Security
- Authorization
- Authorization in SQL

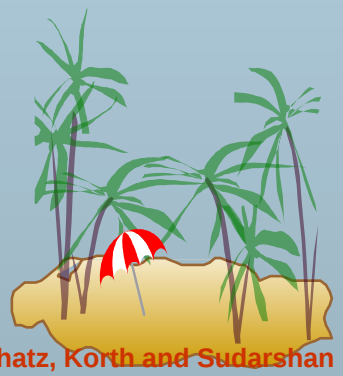


Security

- **Security** - protection from malicious attempts to steal or modify data.

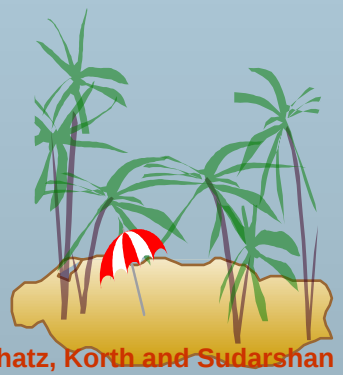
  To protect the database, we must take security measures at several levels:

  - Database system level
    - Authentication and authorization mechanisms to allow specific users access only to required data
    - We concentrate on authorization in the rest of this chapter
  - Operating system level
    - Operating system super-users can do anything they want to the database! Good operating system level security is required.
  - Network level: must use encryption to prevent
    - Eavesdropping (unauthorized reading of messages)
    - Masquerading (pretending to be an authorized user or sending messages supposedly from authorized users)



Security (Cont.)

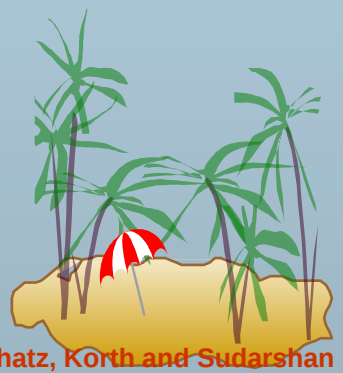
- Physical level
  - Physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed
  - Computers must also be protected from floods, fire, etc.
- Human level
  - Users must be screened to ensure that an authorized users do not give access to intruders
  - Users should be trained on password selection and secrecy



Authorization

Forms of authorization on parts of the database:

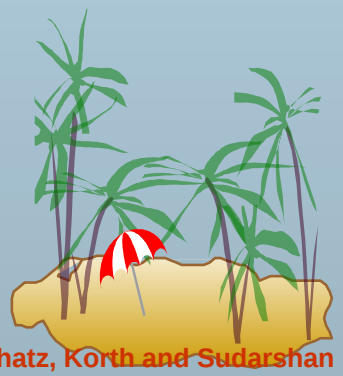
- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data



Authorization (Cont.)

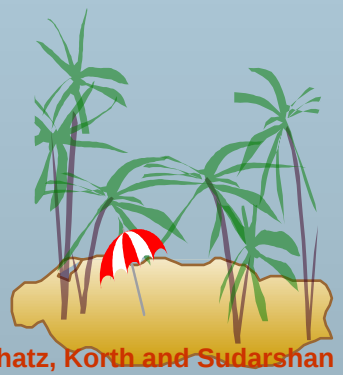
Forms of authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or deletion of attributes in a relation.
- **Drop authorization** - allows deletion of relations.



Authorization and Views

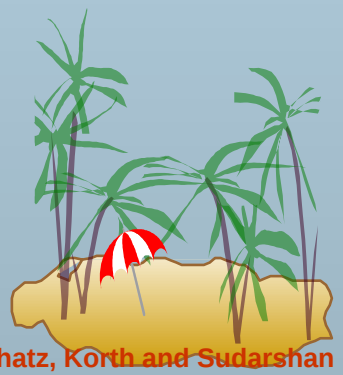
- Users can be given authorization on views, without being given any authorization on the relations used in the view definition
- Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job
- A combination or relational-level security and view-level security can be used to limit a user's access to precisely the data that user needs.



View Example

- Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.
  - ★ Approach: Deny direct access to the *loan* relation, but grant access to the view *cust-loan*, which consists only of the names of customers and the branches at which they have a loan.
  - ★ The *cust-loan* view is defined in SQL as follows:

```
create view cust-loan as
    select branchname, customer-name
    from   borrower, loan
    where borrower.loan-number = loan.loan-number
```

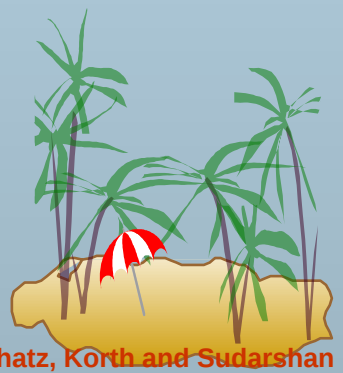


View Example (Cont.)

- The clerk is authorized to see the result of the query:

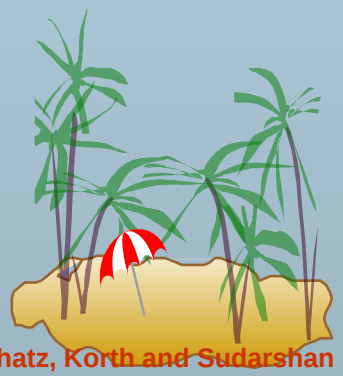
  select *
  from *cust-loan*

- When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*.
- Authorization must be checked on the clerk's query before query processing begins.



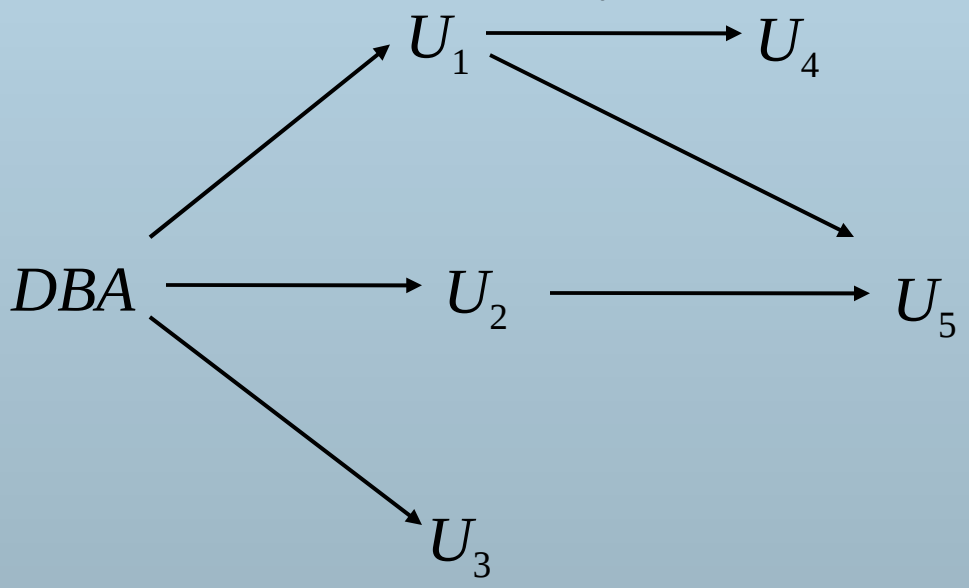
Authorization on Views

- Creation of view does not require **resources** authorization since no real relation is being created
- The creator of a view gets only those privileges that provide no additional authorization beyond that he already had.
- E.g. if creator of view *cust-loan* had only **read** authorization on *borrower* and *loan*, he gets only **read** authorization on *cust-loan*



Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j.



Authorization Grant Graph

- *Requirement*: All edges in an authorization graph must be part of some path originating with the database administrator
- If DBA revokes grant from U_1:
  - ★ Grant must be revoked from U_4 since U_1 no longer has authorization
  - ★ Grant must not be revoked from U_5 since U_5 has another authorization path from DBA through U_2
- Must prevent cycles of grants with no path from the root:
  - ★ DBA grants authorization to U_7
  - ★ U7 grants authorization to U_8
  - ★ U8 grants authorization to U_7
  - ★ DBA revokes authorization from U_7
- Must revoke grant U_7 to U_8 and from U_8 to U_7 since there is no path from DBA to U_7 or to U_8 anymore.

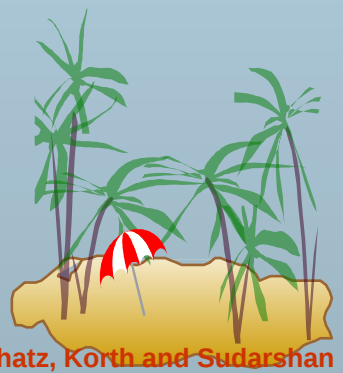
Security Specification in SQL

- The grant statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> to <user list>

- <user list> is:
  - a user-id
  - *public*, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

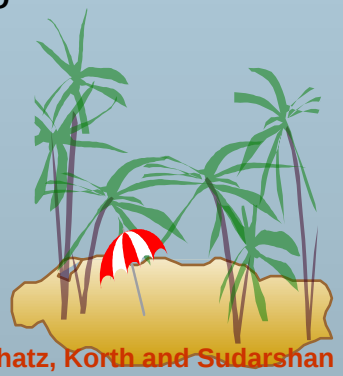


Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - ★ Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

    grant select on *branch* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **references**: ability to declare foreign keys when creating relations.

- **all privileges**: used as a short form for all the allowable privileges

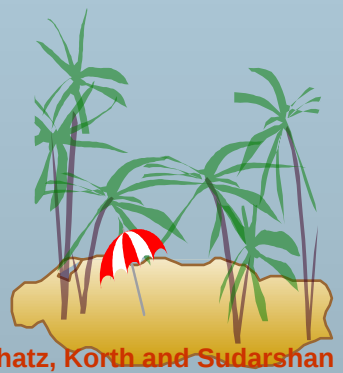


Privilege To Grant Privileges

- **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.
  - ★ Example:

grant select on *branch* **to** U_1 **with grant option**

gives U_1 the **select** privileges on branch and allows U_1 to grant this privilege to others



Roles

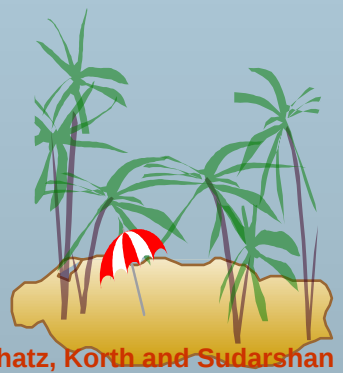
- Roles permit common privileges for a class of users can be specified just once by creating a corresponding “role”
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles
- SQL:1999 supports roles

create role *teller*
create role *manager*

grant select on *branch* **to** *teller*
grant update (*balance***) on** *account* **to** *teller*
grant all privileges on *account* **to** *manager*

grant *teller* **to** *manager*

grant *teller* **to** *alice, bob*
grant *manager* **to** *avi*



Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

revoke<privilege list>

on <relation name or view name> **from** <user list> [**restrict**|**cascade**]

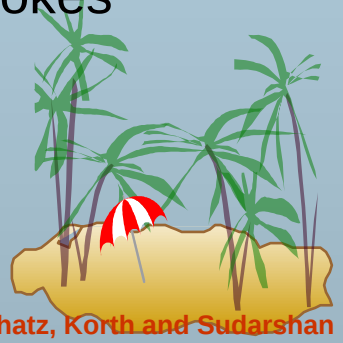
- Example:

revoke select on *branch* **from** U_1, U_2, U_3 **cascade**

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**.
- We can prevent cascading by specifying **restrict**:

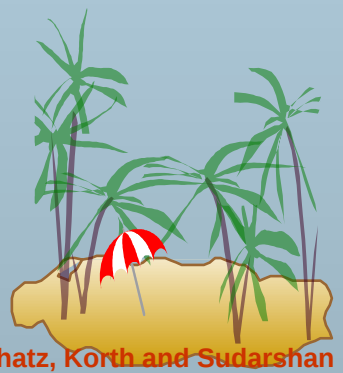
  revoke select on *branch* **from** U_1, U_2, U_3 **restrict**

  With **restrict**, the **revoke** command fails if cascading revokes are required.



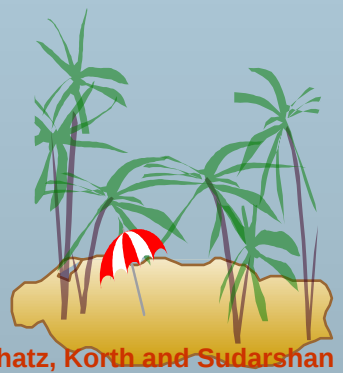
Revoking Authorization in SQL (Cont.)

- <privilege-list> may be **all to** revoke all privileges the revokee may hold.
- If <revokee-list> includes **public** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.



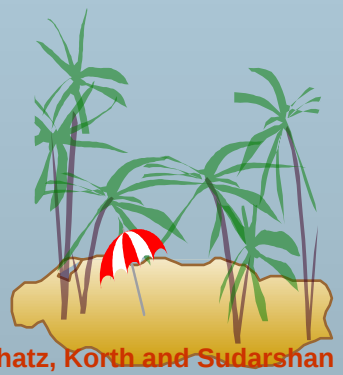
Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
  - ★ E.g. we cannot restrict students to see only (the tuples storing) their own grades
- All end-users of an application (such as a web application) may be mapped to a single database user
- The task of authorization in above cases falls on the application program, with no support from SQL
  - ★ Authorization must be done in application code, and may be dispersed all over an application
  - ★ Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code



Encryption

- Data may be *encrypted* when database authorization provisions do not offer sufficient protection.
- Properties of good encryption technique:
  - ★ Relatively simple for authorized users to encrypt and decrypt data.
  - ★ Encryption scheme depends not on the secrecy of the algorithm but on the secrecy of a parameter of the algorithm called the encryption key.
  - ★ Extremely difficult for an intruder to determine the encryption key.



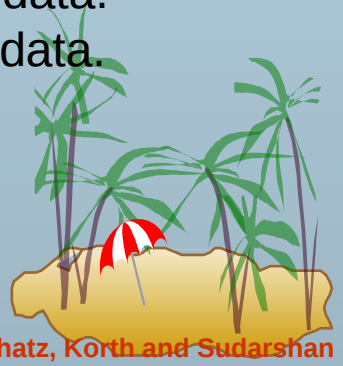
Encryption (Cont.)

- *Data Encryption Standard* (DES) substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorized users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.
- Advanced Encryption Standard (AES) is a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys
- *Public-key encryption* is based on each user having two keys:
  - ★ *public key* – publicly published key used to encrypt data, but cannot be used to decrypt data
  - ★ *private key* -- key known only to individual user, and used to decrypt data. Need not be transmitted to the site doing encryption.

  Encryption scheme is such that it is impossible or extremely hard to decrypt data given only the public key.
- The RSA public-key encryption scheme is based on the hardness of factoring a very large number (100's of digits) into its prime components.

Authentication

- Password based authentication is widely used, but is susceptible to sniffing on a network
- **Challenge-response** systems avoid transmission of passwords
  - ★ DB sends a (randomly generated) challenge string to user
  - ★ User encrypts string and returns result.
  - ★ DB verifies identity by decrypting result
  - ★ Can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back
- **Digital signatures** are used to verify authenticity of data
  - ★ E.g. use private key (in reverse) to encrypt data, and anyone can verify authenticity by using public key (in reverse) to decrypt data. Only holder of private key could have created the encrypted data.
  - ★ Digital signatures also help ensure **nonrepudiation:** sender cannot later claim to have not created the data

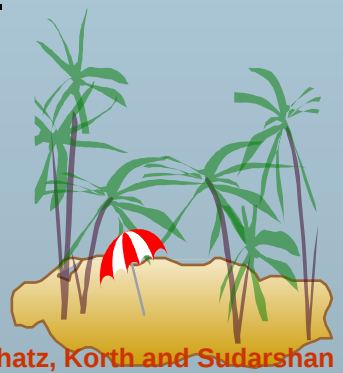


Physical Level Security

- Protection of equipment from floods, power failure, etc.
- Protection of disks from theft, erasure, physical damage, etc.
- Protection of network and terminal cables from wiretaps non-invasive electronic eavesdropping, physical damage, etc.

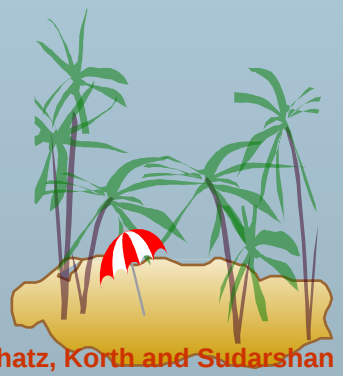
Solutions:

- Replicated hardware:
  - ★ mirrored disks, dual busses, etc.
  - ★ multiple access paths between every pair of devises
- Physical security: locks,police, etc.
- Software techniques to detect physical security breaches.



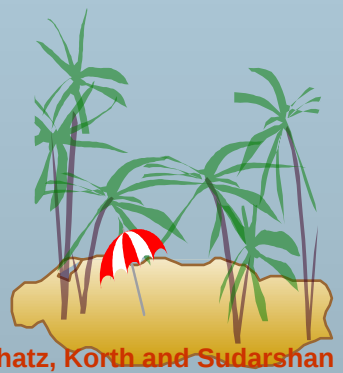
Human Level Security

- Protection from stolen passwords, sabotage, etc.
- Primarily a management problem:
  - ★ Frequent change of passwords
  - ★ Use of “non-guessable” passwords
  - ★ Log all invalid access attempts
  - ★ Data audits
  - ★ Careful hiring practices



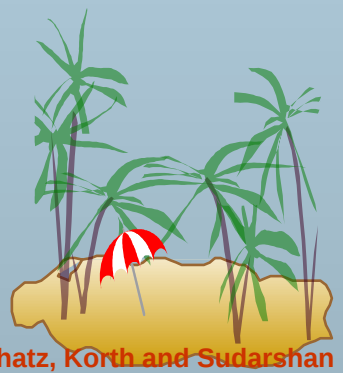
Operating System Level Security

- Protection from invalid logins
- File-level access protection (often not very helpful for database security)
- Protection from improper use of “superuser” authority.
- Protection from improper use of privileged machine intructions.



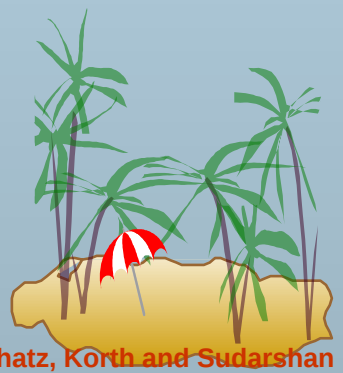
Network-Level Security

- Each site must ensure that it communicate with trusted sites (not intruders).
- Links must be protected from theft or modification of messages
- Mechanisms:
  - ★ Identification protocol (password-based),
  - ★ Cryptography.



Database-Level Security

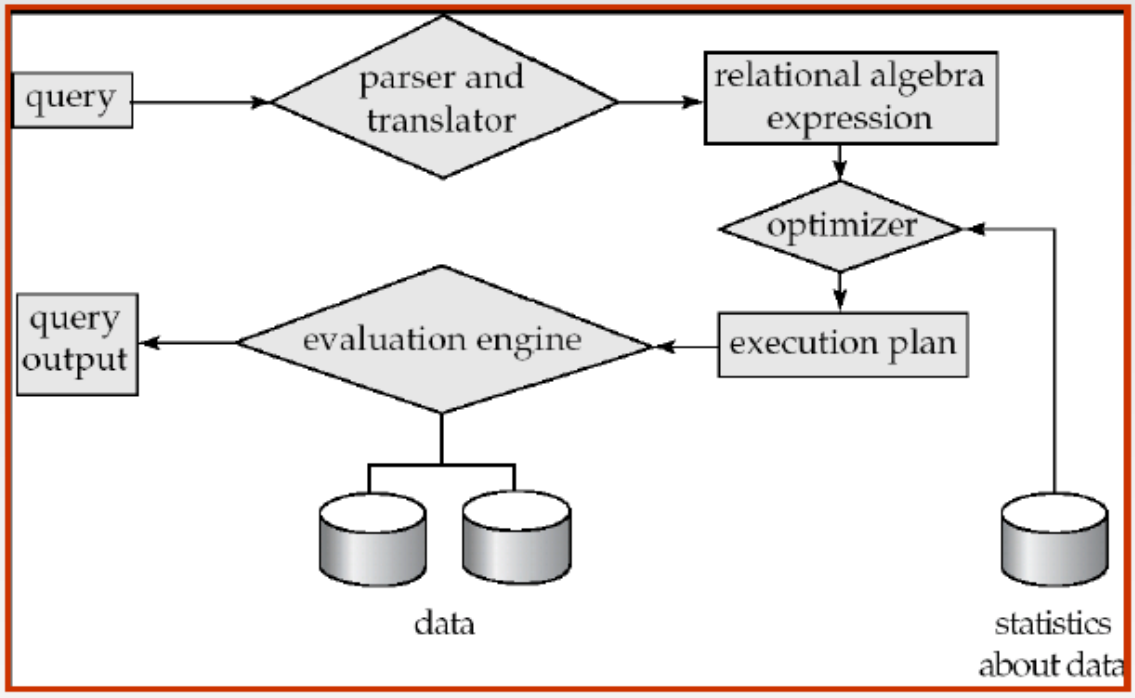
- Assume security at network, operating system, human, and physical levels.
- Database specific issues:
  - ★ each user may have authority to read only part of the data and to write only part of the data.
  - ★ User authority may correspond to entire files or relations, but it may also correspond only to parts of files or relations.
- Local autonomy suggests site-level authorization control in a distributed database.
- Global control suggests centralized control.



Chapter 7: Query processing

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



- **Parsing and translation:** The first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.
- **Evaluation:** A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations

that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**.

As an illustration, consider the query

select *balance* **from** *account* **where** *balance* < 2500

Figure below illustrates an evaluation plan for our example query,

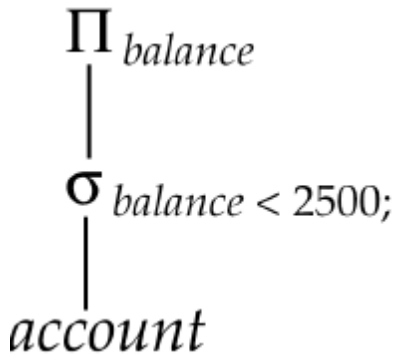


Fig: A query-evaluation plan.

The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost. Cost is estimated using statistical information from the database catalog e.g. number of tuples in each relation, size of tuples, etc. The different evaluation plans for a given query can have different costs.

Measures of Query Cost

Cost is generally measured as total elapsed time for answering query. The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication.

Typically disk access is the predominant cost, and is also relatively easy to estimate. It is measured by taking into account

Number of seeks * average-seek-cost
+ Number of blocks read * average-block-read-cost
+ Number of blocks written * average-block-write-cost

For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures

- t_T – time to transfer one block
- t_S – time for one seek
- Cost for b block transfers plus S seeks

$b * t_T + S * t_S$

We ignore CPU costs for simplicity but Real systems do take CPU cost into account. We do not include cost to writing output to disk in our cost formulae.

Query Optimization:

It is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play**.** One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute.

Consider relational-algebra expression for the query "Find the names of all customers who have an account at any branch located in Brooklyn."

$$\Pi_{customer\text{-}name}\left(\sigma_{branch-city="Brooklyn"}\left(branch \bowtie \left(account \bowtie depositor\right)\right)\right)$$

This expression constructs a large intermediate relation, *branch* ⋈ *account* ⋈ *depositor*.

However, we are interested in only a few tuples of this relation (those pertaining to branches located in Brooklyn), and in only one of the six attributes of this relation. Since we are concerned with only those tuples in the *branch* relation that pertain to branches located in Brooklyn, we do not need to consider those tuples that do not have *branch-city* = "Brooklyn". By reducing the number of tuples of the *branch* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression

$$\Pi_{customer\text{-}name}\left(\left(\sigma_{branch\text{-}city="Brooklyn"}\left(branch\right)\right) \bowtie \left(account \bowtie depositor\right)\right)$$

which is equivalent to our original algebra expression, but which generates smaller intermediate relations. Figure below depicts the initial and transformed expressions.

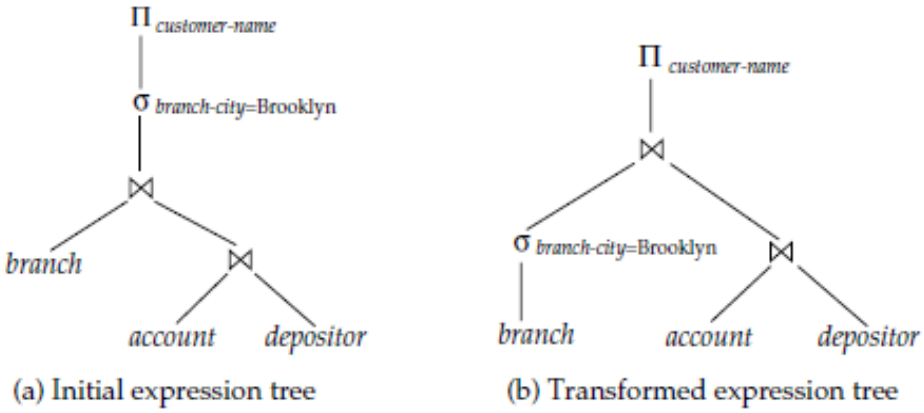


Fig: Equivalent Expressions

- Cost difference between evaluation plans for a query can be enormous e.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - ✓ Statistical information about relations. Examples: number of tuples, number of distinct values for an attribute
- Statistics estimation for intermediate results
  - ✓ to compute cost of complex expressions

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every legal database instance
  - ✓ Note: order of tuples is irrelevant
- An **equivalence rule** says that expressions of two forms are equivalent i.e can replace expression of first form by second, or vice versa.

Equivalence Rules

We now list a number of general equivalence rules on relational-algebra expressions. We use ϑ, $\vartheta 1$, $\vartheta 2$, and so on to denote predicates, $L1$, $L2$, $L3$, and so on to denote lists of attributes, and
E, $E1$, $E2$, and so on to denote relational-algebra expressions.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are **commutative**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the final operations in a sequence of projection operations are needed, the others can be omitted. This transformation can also be referred to as a cascade of Π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   - **a.** $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
     This expression is just the definition of the theta join.
   - **b.** $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta-join operations are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. **a.** Natural-join operations are **associative**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

   b. Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

   where θ_2 involves attributes from only E_2 and E_3. Any of these conditions may be empty; hence, it follows that the Cartesian product ($\times$) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

7. The selection operation distributes over the theta-join operation under the following two conditions:
   - **a.** It distributes when all the attributes in selection condition θ_0 involve only the attributes of one of the expressions (say, E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

   - **b.** It distributes when selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta-join operation under the following conditions.

   a. Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   b. Consider a join $E_1 \bowtie_\theta E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   Set difference is not commutative.

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over the union, intersection, and set–difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

    Similarly, the preceding equivalence, with $-$ replaced with either $\cup$ or $\cap$, also holds. Further,

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

    The preceding equivalence, with $-$ replaced by $\cap$, also holds, but does not hold if $-$ is replaced by $\cup$.

12. The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Example of Transformations: Pushing Selections

We now illustrate the use of the equivalence rules. We use our bank example with the relation schemas:

$$\begin{aligned} &\textit{Branch-schema} = (\textit{branch-name}, \textit{branch-city}, \textit{assets}) \\ &\textit{Account-schema} = (\textit{account-number}, \textit{branch-name}, \textit{balance}) \\ &\textit{Depositor-schema} = (\textit{customer-name}, \textit{account-number}) \end{aligned}$$

The relations *branch*, *account*, and *depositor* are instances of these schemas.

In our example in Section 14.1, the expression

$$\Pi_{\textit{customer-name}}(\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})))$$

was transformed into the following expression,

$$\Pi_{\textit{customer-name}}((\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\textit{branch})) \bowtie (\textit{account} \bowtie \textit{depositor}))$$

which is equivalent to our original algebra expression, but generates smaller intermediate relations. We can carry out this transformation by using rule 7.a. Remember that the rule merely says that the two expressions are equivalent; it does not say that one is better than the other.

Multiple equivalence rules can be used, one after the other, on a query or on parts of the query. As an illustration, suppose that we modify our original query to restrict attention to customers who have a balance over $1000. The new relational-algebra query is

$$\Pi_{\textit{customer-name}}(\sigma_{\textit{branch-city} = \text{"Brooklyn"} \wedge \textit{balance} > 1000}(\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})))$$

We cannot apply the selection predicate directly to the *branch* relation, since the predicate involves attributes of both the *branch* and *account* relation. However, we can first apply rule 6.a (associativity of natural join) to transform the join $\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})$ into $(\textit{branch} \bowtie \textit{account}) \bowtie \textit{depositor}$:

$$\Pi_{\textit{customer-name}}(\sigma_{\textit{branch-city} = \text{"Brooklyn"} \wedge \textit{balance} > 1000}((\textit{branch} \bowtie \textit{account}) \bowtie \textit{depositor}))$$

Then, using rule 7.a, we can rewrite our query as

$$\Pi_{\textit{customer-name}}((\sigma_{\textit{branch-city} = \text{"Brooklyn"} \wedge \textit{balance} > 1000}(\textit{branch} \bowtie \textit{account})) \bowtie \textit{depositor})$$

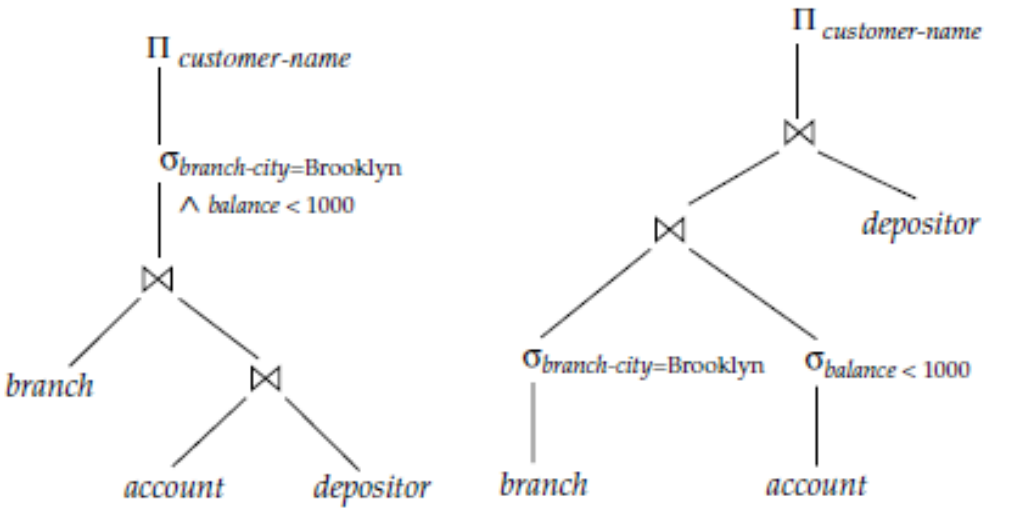
Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections, to get the following subexpression:

$$\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\sigma_{\textit{balance} > 1000}(\textit{branch} \bowtie \textit{account}))$$

Both of the preceding expressions select tuples with *branch-city* = "Brooklyn" and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the "perform selections early" rule, resulting in the subexpression

$$\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\textit{branch}) \bowtie \sigma_{\textit{balance} > 1000}(\textit{account})$$

Figure 14.3 depicts the initial expression and the final expression after all these transformations. We could equally well have used rule 7.b to get the final expression directly, without using rule 1 to break the selection into two selections. In fact, rule 7.b can itself be derived from rules 1 and 7.a



(a) Initial expression tree (b) Tree after multiple transformations

Figure 14.3 Multiple transformations.

Transformation Example: Pushing Projections:

Now consider the following form of our example query:

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch) \bowtie account) \bowtie depositor)$$

When we compute the subexpression

$$(\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch) \bowtie account)$$

we obtain a relation whose schema is

(*branch-name, branch-city, assets, account-number, balance*)

We can eliminate several attributes from the schema, by pushing projections based on equivalence rules 8.a and 8.b. The only attributes that we must retain are those that either appear in the result of the query or are needed to process subsequent operations. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, we reduce the size of the intermediate result. In our example, the only attribute we need from the join of *branch* and *account* is *account-number*. Therefore, we can modify the expression to

$$\Pi_{customer\text{-}name}\ (\ (\Pi_{account\text{-}number}\ ((\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch)) \bowtie account)) \bowtie depositor)$$

The projection $\Pi_{account\text{-}number}$ reduces the size of the intermediate join results.

Solved Examples:

Q. Show how to derive the following equivalences by a sequence of Transformations using the equivalence rules.

a. $\sigma\theta1\theta2\theta3(E) = \sigma\theta1\ (\sigma\theta2\ (\sigma\theta3\ (E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2

Answer:

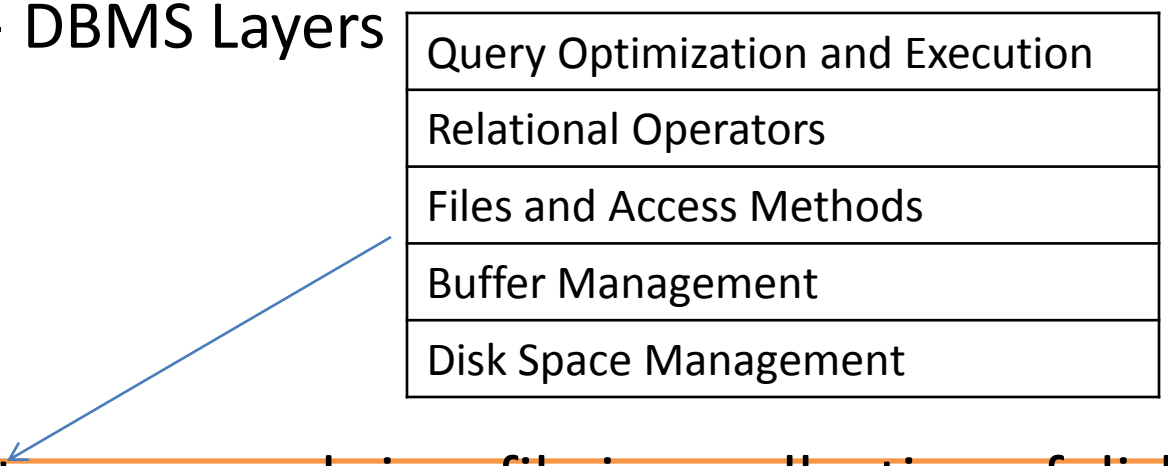
a. Using rule 1, $\sigma\theta1\theta2\theta3(E)$ becomes $\sigma\theta1\ (\sigma\theta2\theta3(E))$. On applying rule 1 again, we get $\sigma\theta1\ (\sigma\theta2\ (\sigma\theta3\ (E)))$.

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2)$ on applying rule 1 becomes $\sigma_{\theta_1}(\sigma_{\theta_2}(E_1 \bowtie_{\theta_3} E_2))$. This on applying rule 7.a becomes $\sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$.

Chapter 8:
File Organization and Indexing

File Organization

- File organization: is a method of arranging the records in a file when the file is stored on disk. A relation is typically stored as a file of records.
  - DBMS Layers



Stores records in a file in a collection of disk pages.
Keeps track of pages allocated to each file.
Tracks available space within pages allocated to the file.

Data on External Storage

- A DBMS stores vast amounts of data and the data has to be preserved across program executions.
- Therefore, data is stored on external storage and fetched into main memory as needed for processing.
- The unit of information that is read and written to a disk is Page.
- Higher layer of DBMS views this pages as unified files and can read or write records to these files.

Classification of Physical Storage Media

Several types of data storage exist in most computer system

These storage are classified by

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
  - data loss on power failure or system crash
  - physical failure of the storage device
- Can differentiate storage into:
  - volatile storage: loses contents when power is switched off
  - non-volatile storage: Contents persist even when power is switched off.

Storage Media

Among the media typically available are these:

- Cache:
  - The fastest and most costly form of storage.
- Main Memory:
  - The general purpose machine instructions operate in main memory
  - used for data that are available to be operated.
  - Is volatile in nature.
- Flash Memory:
  - Is non-volatile in nature
  - Reading data from flash memory is at speed of that of main memory

Storage Media

- Flash Memory:
  - Writing data to flash memory is slower
  - More expensive than disks
  - Used for applications with read workloads that require fast random accesses.
  - Mostly used in embedded systems like in hand held devices and other digital electronic devices.
- Magnetic Disk :
  - Secondary storage, used for long term storage of data.
  - Data should be moved to main memory before in use.
  - It is non volatile in nature.
  - Direct Memory Access.

Storage Media

- Optical Storage:
  - Random access of data
  - Used as secondary access of data for long term .
  - Mostly found in compact disk (CD) and digital video disk(DVD).
  - Data are stored optically on a disk and read by a laser.
  - Capable of storing high data in Gigabytes.
  - Found both in read only and read write form as CD-R, CD-RW,DVD-R,DVD-RW
- Magnetic Tape :
  - Is referred as sequential access storage.
  - Primarily used for backup and archival data .
  - Cheaper than disks but also access to data is slower.

Storage Media

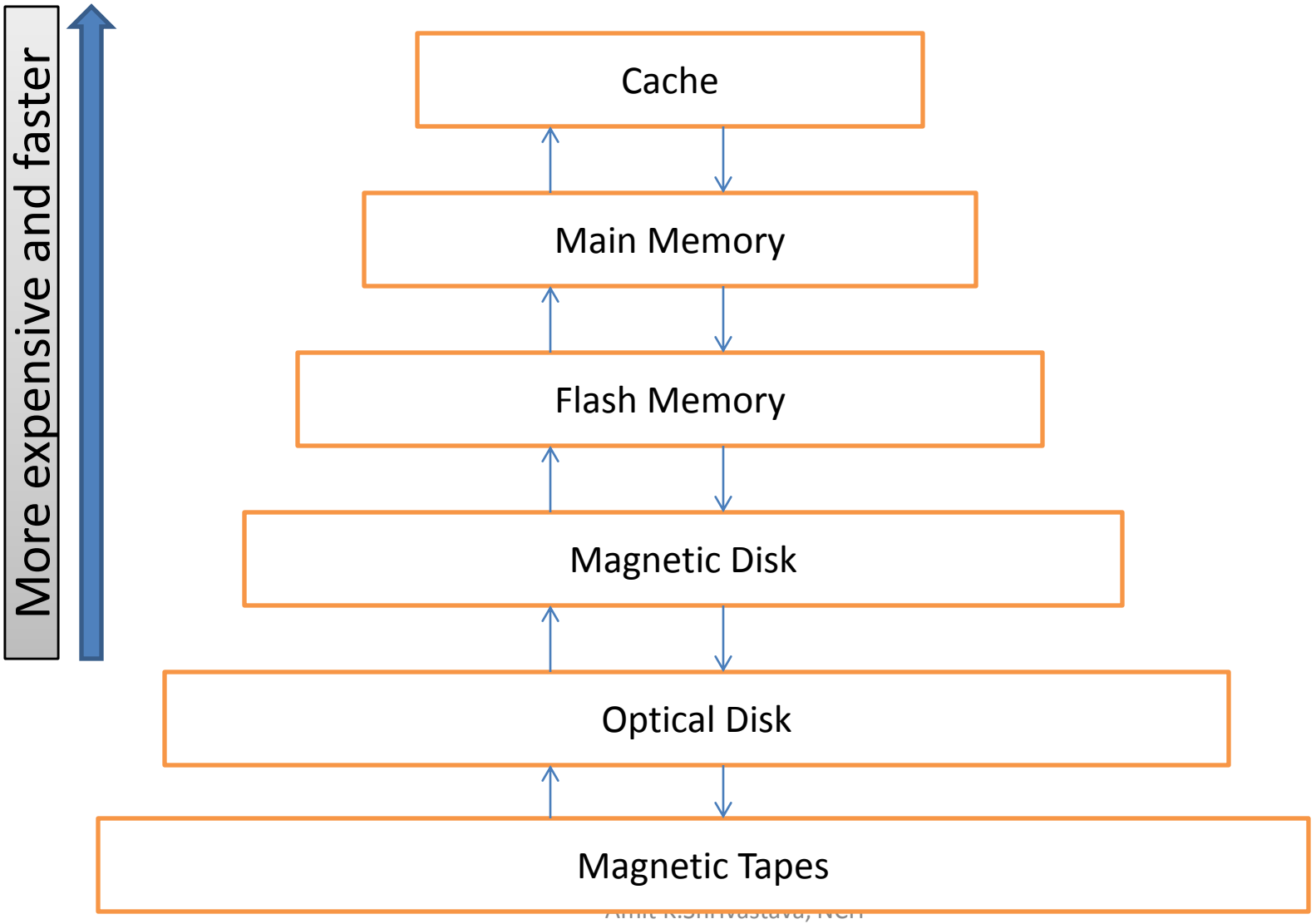


Fig.: Storage device Hierarchy

Buffer Manager

- Subsystem that is responsible for loading pages from external storage to the main memory (buffer pool) when needed.
- File & Index layers make calls to the buffer manager.
- If block is already in buffer, BM passes address of block in main memory.
- Else, BM first allocates space in the buffer for block.
- BM reads in requested block from disk to buffer and passes address of block in main memory.
- Idea is to Keep as many blocks(pages) in memory as possible to reduce disk accesses.
- Replacement Policies
  - LRU(Least-Recently-Used pages … are discarded => the oldest are discarded first)
  - MRU(Most-Recently-Used, the newest are discarded first)
  - LFU(Least-Frequently-Used)

File Organization

- Method for arranging a collection of records and supporting the concept of a file.
- A file is organized logically as a sequence of records. These records are mapped into the disk blocks.
- Each file is also logically partitioned into fixed length storage units called blocks, which are the units of both storage allocation and data transfer.
- Normally the block size range from 4 to 8 kilo bytes by default to Giga Bytes.
- Blocks contains records that is determined by the form of physical data organization being used.
- The size of the records should not be greater than block considering the fact that the field might be of image and it can varies in size dramatically.
- The another fact is that the entire record should be contained in one block rather than divided into separate blocks. This will benefits in speeding up the data access.
- Records may varies in size in one particular file system, so it is to be addressed.

Fixed Length Records

- Giving the attributes fixed size for an entity in terms of byte.
- For the file student, record may contain id int, name varchar(20), age int. For each record the space required is 24 bytes.
- **Problems**
  - The file records are of the same record type, but one or more of the fields are of varying size.
  - The file records are of the same record type but one or more of the fields may have multiple values for individual's records.
  - The file records are of the same records type but one or more of the fields are optional.
  - But it is very hard to occupy the block size of multiple of 24 bytes. So, it may be possible that when the block size obtain its max limit, it might be possible that the information of records may shift to another block.
  - While deleting the records from file, it should be either marked as deleted or should be occupied by next new record.

Fixed Length Records

- **Solutions**
  - The records should be entered into the block after the block size computation (dividing block size with the record size), by leaving the remaining block size unused.
  - While deleting record, we could move the record that came after it into the space formerly occupied by the deleted record and so on, until every record following the deleted record has been moved ahead. Or it might be easier to move the final record of the file into the space occupied by the deleted record.
  - The file header is used for storing the information of deleted records as well as available records. While inserting the new record, if the file contains the deleted spaces it will insert the record in that position and so on, if it does not contain any deleted position the record is placed at the end of the file

Fixed Length Records

Record 0	A-102	Kalimati	4000
Record 1	A-201	Patan	5000
Record 2	A-302	Bhaktapur	6000
Record 3	A-402	kalanki	4000
Record 4	A-103	kalimati	5000
Record 5	A-502	kirtipur	3000
Record 6	A-105	kalimati	5000
Record 7	A-202	patan	4000
Record 8	A-403	kalanki	4000

Record 0	A-102	Kalimati	4000
Record 1	A-201	Patan	5000
Record 3	A-402	kalanki	4000
Record 4	A-103	kalimati	5000
Record 5	A-502	kirtipur	3000
Record 6	A-105	kalimati	5000
Record 7	A-202	patan	4000
Record 8	A-403	kalanki	4000

Record 2 deleted and all records moved up

Fixed Length Records

Record 0	A-102	Kalimati	4000
Record 1	A-201	Patan	5000
Record 8	A-403	kalanki	4000
Record 3	A-402	kalanki	4000
Record 4	A-103	kalimati	5000
Record 5	A-502	kirtipur	3000
Record 6	A-105	kalimati	5000
Record 7	A-202	patan	4000

Record 2 deleted and final record moved

header			
Record 0	A-102	Kalimati	4000
Record 1			
Record 2	A-302	Bhaktapur	6000
Record 3	A-402	kalanki	4000
Record 4			
Record 5	A-502	kirtipur	3000
Record 6			
Record 7	A-202	patan	4000
Record 8	A-403	kalanki	4000

Free list after deletion of records 1,4,6

Variable Length Records

- It is used for
  - storing multiple record type in a file
  - record type that allow variable lengths for one or more fields
  - record types that allow repeating fields, such as arrays.
- Byte String Representation:
  - Attach a special *end of record* (⊥) symbol to the end of each record.
  - Store each record as a string of consecutive bytes.
  - Disadvantages:
    - Not easy to reuse space occupied formerly by a deleted record.
    - No space for records to grow longer.

Variable Length Records

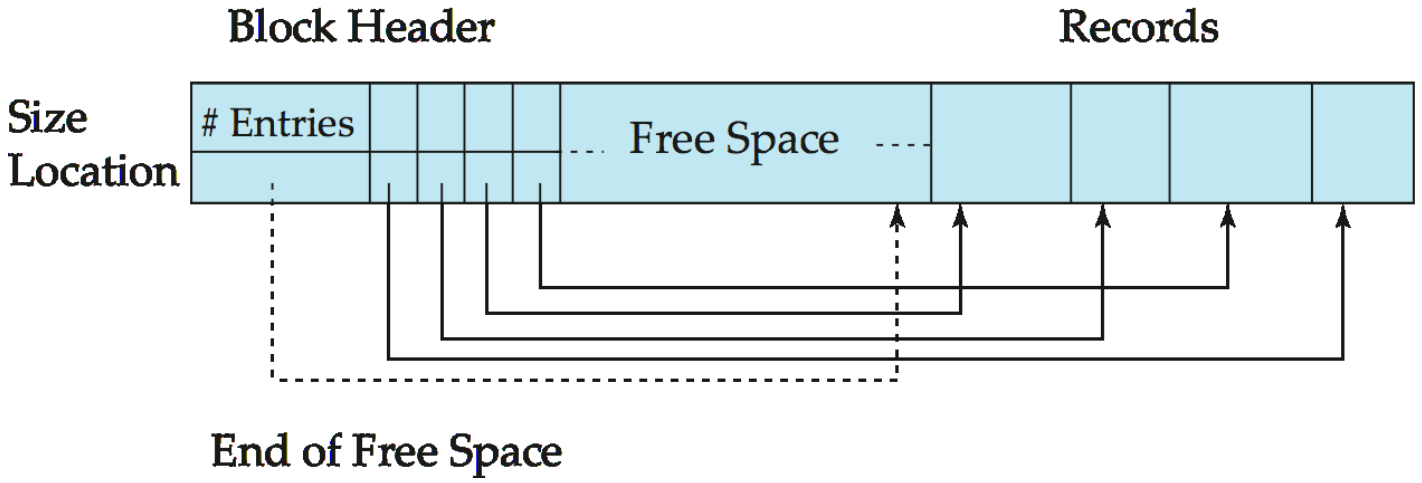
- Byte String Representation:

0	Kalimati	A-102	4000	A-103	5000	A-105	5000	
1	Patan	A-201	5000	A-202	4000			
2	Bhaktapur	A-303	6000					
3	Kalanki	A-402	4000	A-403	4000			
4	Kirtipur	A-502	3000					

- Slotted page structure:
  - Header consists of
    - number of record entries
    - end of free space in the block
    - location and size of each record

Variable Length Records

- Slotted page structure:
  - Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
  - Pointers should not point directly to record — instead they should point to the entry for the record in header.



Variable Length Records

- Fixed Length Representation:
  - Two ways
    - Reserved Space
    - List Representation

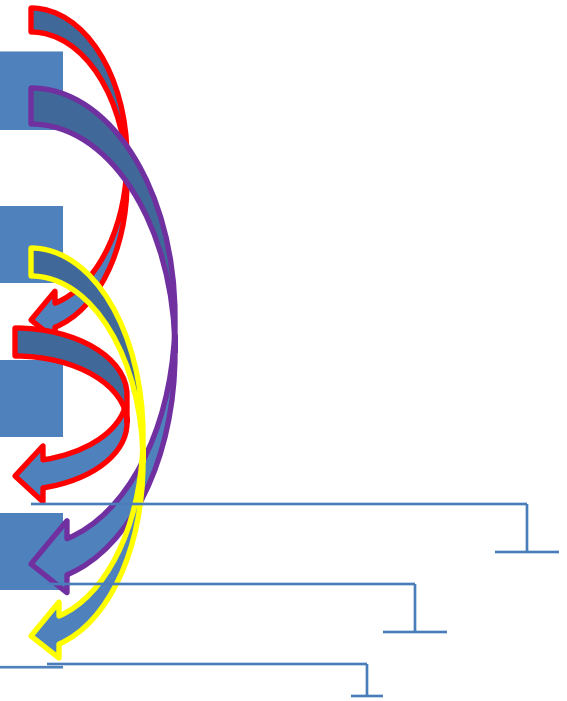
0	Kalimati	A-102	4000	A-103	5000	A-105	5000
1	Patan	A-201	5000	A-202	4000		
2	Bhaktapur	A-303	6000	⊥	⊥	⊥	⊥
3	Kalanki	A-402	4000	A-403	4000		
4	Kirtipur	A-502	3000	⊥	⊥	⊥	⊥

Reserved Space Method

Variable Length Records

- Fixed Length Representation:

Record 0	Kalimati	A-102	4000
Record 1	Patan	A-201	5000
Record 2	Bhaktapur	A-302	6000
Record 3	kalanki	A-402	4000
Record 4		A-103	5000
Record 5	kirtipur	A-502	3000
Record 6		A-105	5000
Record 7		A-202	4000
Record 8		A-403	4000



Linked List Method

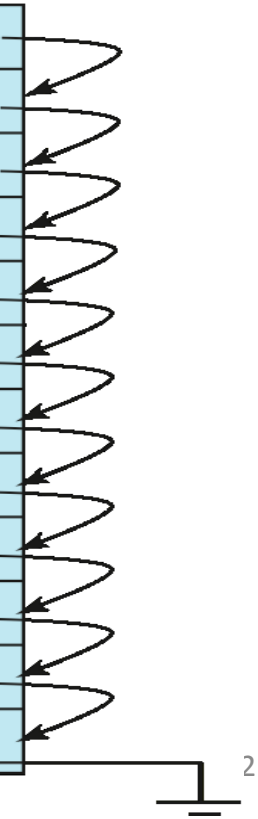
Organization of Records in Files

- **Heap**
  - a record can be placed anywhere in the file where there is space
- **Sequential**
  - store records in sequential order, based on the value of the search key of each record
- **Hashing**
  - a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

Sequential File Organization

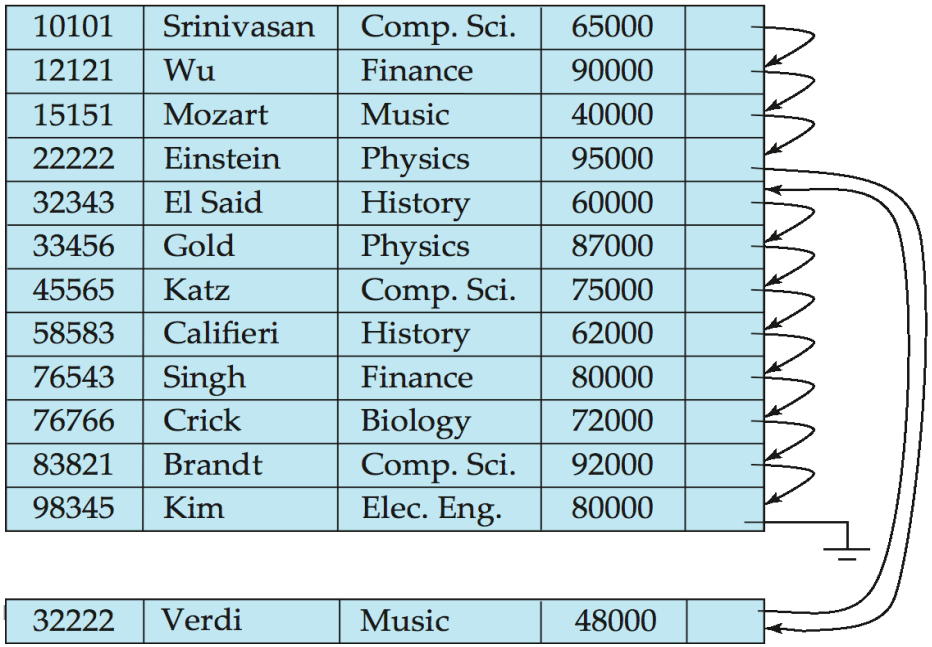
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

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	



Sequential File Organization

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



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	

32222	Verdi	Music	48000	

Data Dictionary Storage

- The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

I. Information about relations
   - names of relations
   - names, types and lengths of attributes of each relation
   - names and definitions of views
   - integrity constraints

II. User and accounting information, including passwords

III. Statistical and descriptive data
   - number of tuples in each relation

IV. Physical file organization information
   - How relation is stored (sequential/hash/…)
   - Physical location of relation

V. Information about indices

Indexing

- Indexing mechanisms are used to speed up access to desired data.
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

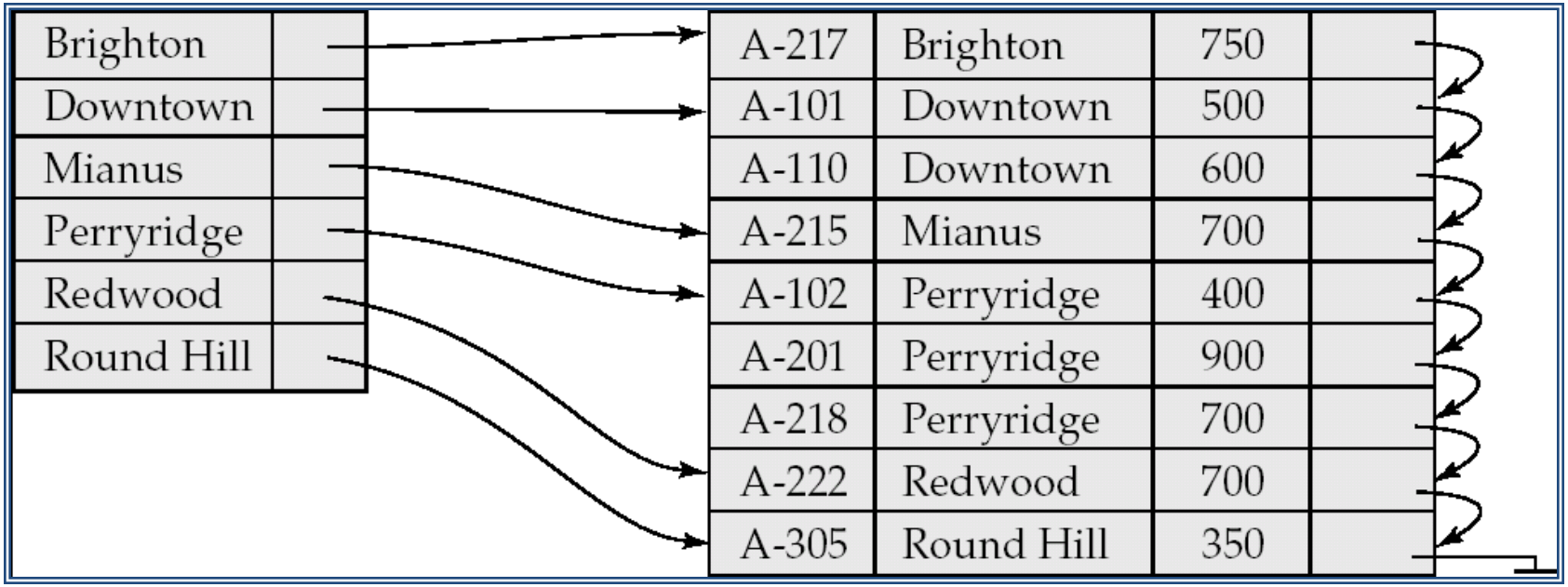
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".
- Index Evaluation Metrics:
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead
  - Access types supported efficiently. E.g.,
    - records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
    - This strongly influences the choice of index, and depends on usage.

1.Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** *in a sequentially ordered file*, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file.
  - Also called non-clustering index**.**
- *Index-sequential file: ordered sequential file with a primary index.*

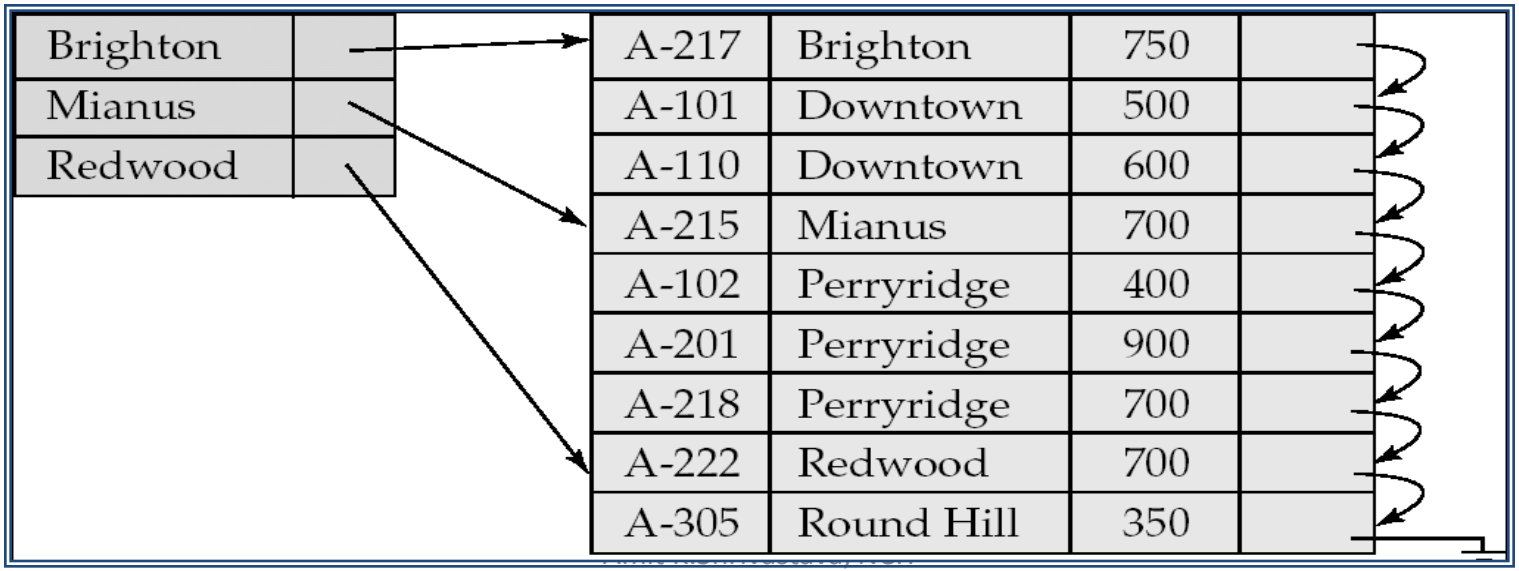
1.1.1Dense index

- Index record appears for every search-key value in the file.
- In dense primary index, the index record contains the search key value and a pointer to the first data record with that search key value.



1.1.2 Sparse Index

- Index records appears for only some search-key values.
- Here also the index record contains the search key value and a pointer to the first data record with that search key value.
- Only applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



1.1.3 Multilevel Indices

- If primary index does not fit in memory, access becomes expensive.
- Solution:
  - treat primary index kept on disk as a sequential file and construct a sparse index on it.
    - outer index – a sparse index of primary index
    - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.
- To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire.
- The pointer points to the block of inner index. We scan this block until we find the record that has the largest search key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking

1.1.4 Index Update

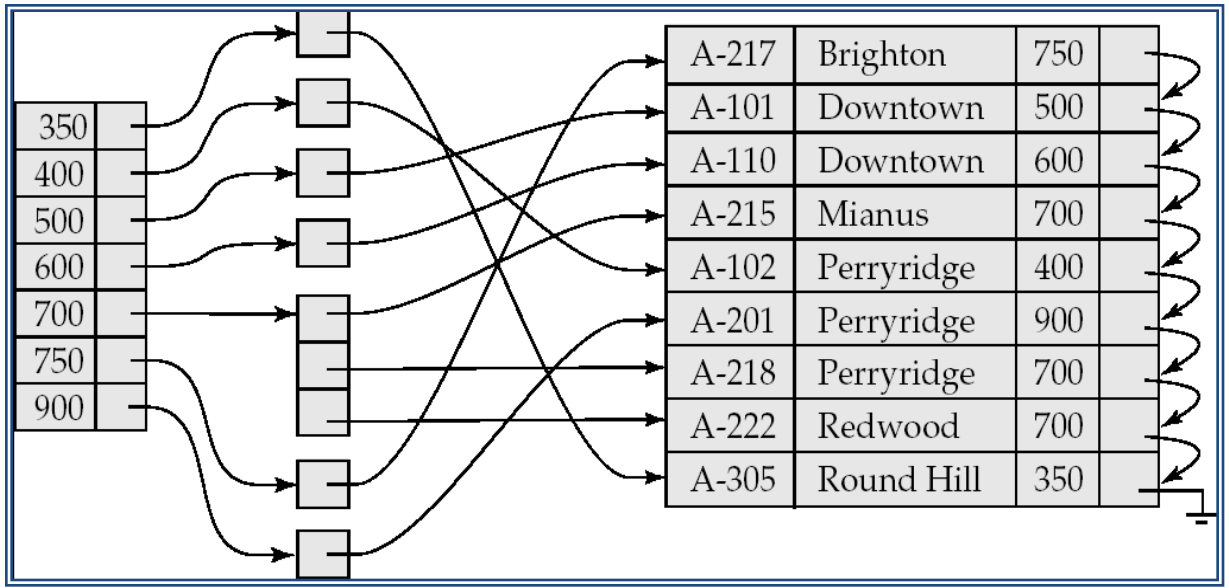
- **Deletion**
  - If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
  - Single-level index deletion:
    - **Dense indices** – deletion of search-key: similar to file record deletion.
    - **Sparse indices** –
      - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
      - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

1.1.4 Index Update

- **Insertion**
  - Single-level index insertion:
    - Perform a lookup using the search-key value appearing in the record to be inserted.
    - **Dense indices** – if the search-key value does not appear in the index, insert it.
    - **Sparse indices** –
      - if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
      - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
  - Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

1.2 Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *account* relation stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense, since the file is not sorted by the search key.



1.2.1 Secondary and Primary Indices

- Indices offer substantial benefits when searching for records, but updating indices imposes overhead on database modification - when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 micro seconds, versus about 100 nanoseconds for memory access

B^+-Tree Index Files

- B^+-tree indices are an alternative to indexed-sequential files.
- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively
- Typical node

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

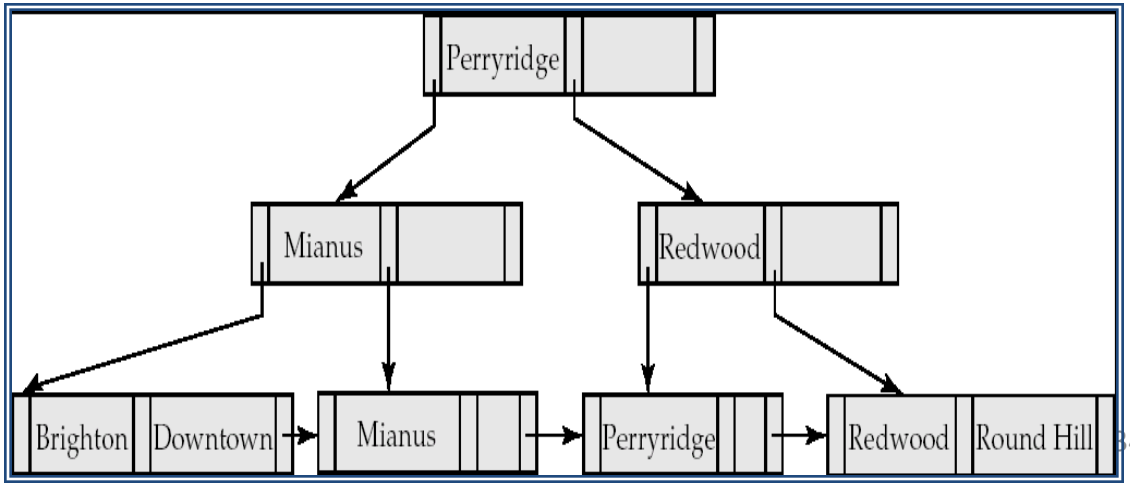
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

- Usually the size of a node is that of a block

B^+-Tree Index Files

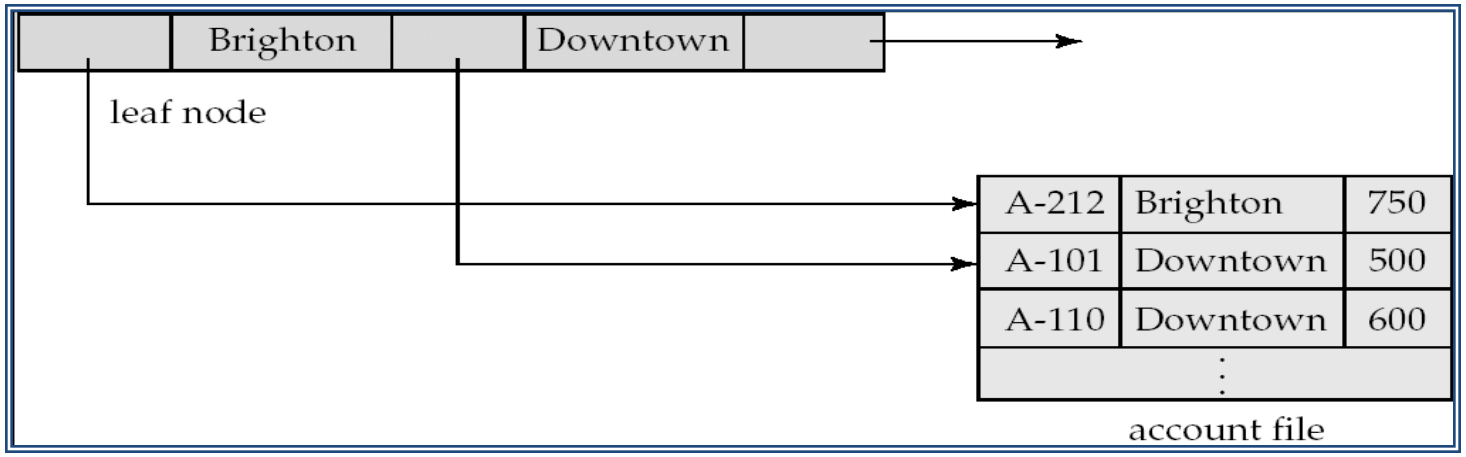
- A B^+-tree is a rooted tree satisfying the following properties:
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
  - A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
  - Special cases:
    - If the root is not a leaf, it has at least 2 children.
    - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.
  - All values must be at leaf node.

B^+-tree for *account* file (n = 3)



Leaf Nodes in B^+-Trees

- Properties of a leaf node:
  - For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
  - If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
  - P_n points to next leaf node in search-key order



Non-Leaf Nodes in B$^+$-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
- *Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and n –1, with n = 5).*
- *Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n/2 \rceil$ and n with n =5).*
- *Root must have at least 2 children.*

B^+-Trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
    - Level below root has at least $2* \lceil n/2 \rceil$ values
    - Next level has at least $2* \lceil n/2 \rceil * \lceil n/2 \rceil$ values
    - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see some details, and more in the book).

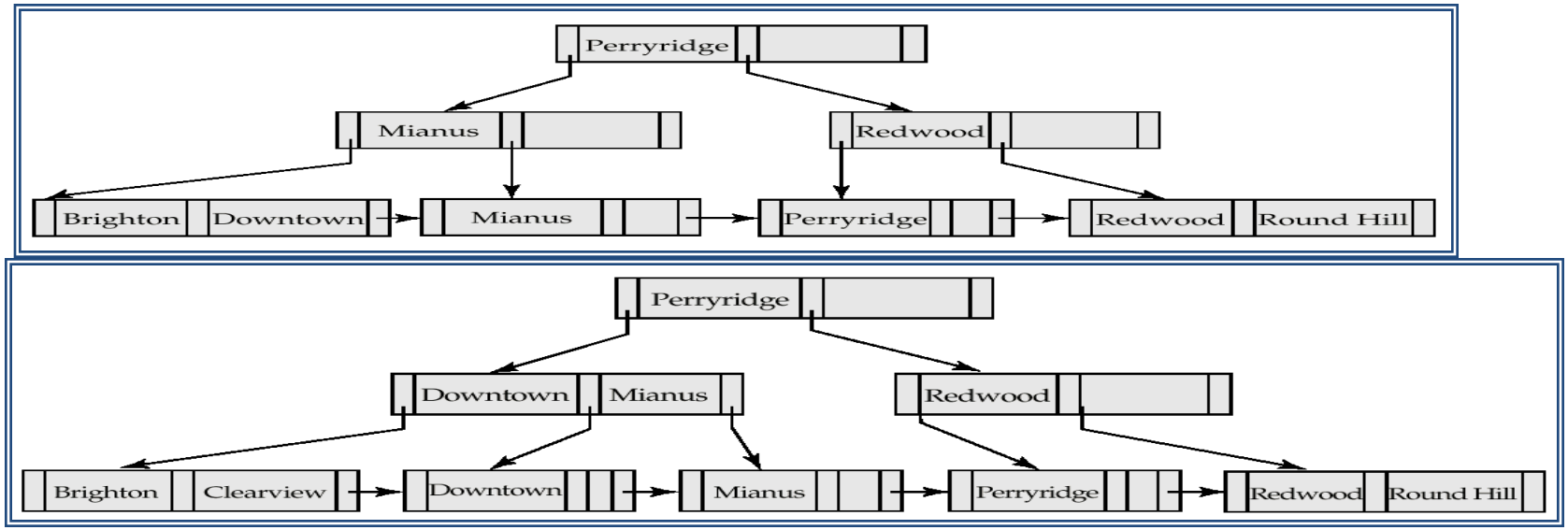
Queries on B^+-Trees

- Find all records with a search-key value of *k.*
  1. *N=root*
  2. Repeat
     1. Examine *N* for the smallest search-key value > *k.*
     2. If such a value exists, assume it is K_i. Then set $N = P_i$
     3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$

     Until *N* is a leaf node
  3. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket.
  4. Else no record with search-key value *k* exists.
- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4Kbytes
  - *n* is typically around 100 (40 bytes per index entry).
- With 1 million search key values and *n* = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
    - I.e. at most 4 accesses to disk blocks are needed
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   1. Add record to the file
   2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
   1. add the record to the main file (and create a bucket if necessary)
   2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   3. Otherwise, split the node (along with the new (key-value, pointer) entry)



B+-Tree before and after insertion of "Clearview"

Updates on B$^+$-Trees: Insertion

- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split.
  - If the parent is full, split it and propagate the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.
- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

Updates on B^+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
  - Not possible to sequentially scan a table by just looking at leafs.
- Typically, advantages of B-Trees do not out weigh disadvantages.
  - In DBMSs B+-Trees are favored.

Hashing

- Hashing is an effective technique to calculate direct location of data record on the disk without using index structure.
- It uses a function, called hash function and generates address when called with search key as parameters. Hash function computes the location of desired data on the disk.
- Hash Organization:
  - **Bucket:** Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.
  - **Hash Function:** A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.
    - Choose hash function that assign search key values to buckets in such a way that distribution is either uniform or random.

Static Hashing

- In static hashing, when a search-key value is provided the hash function always computes the same address.
- For example, if mod-4 hash function is used then it shall generate only 5 values.
- The output address shall always be same for that function. The numbers of buckets provided remain same at all times.
- Operation:
  - **Insertion:** When a record is required to be entered using static hash, the hash function h, computes the bucket address for search key K, where the record will be stored.
    - Bucket address = h(K)
  - **Search:** When a record needs to be retrieved the same hash function can be used to retrieve the address of bucket where the data is stored.
  - **Delete:** This is simply search followed by deletion operation.

Bucket Overflows

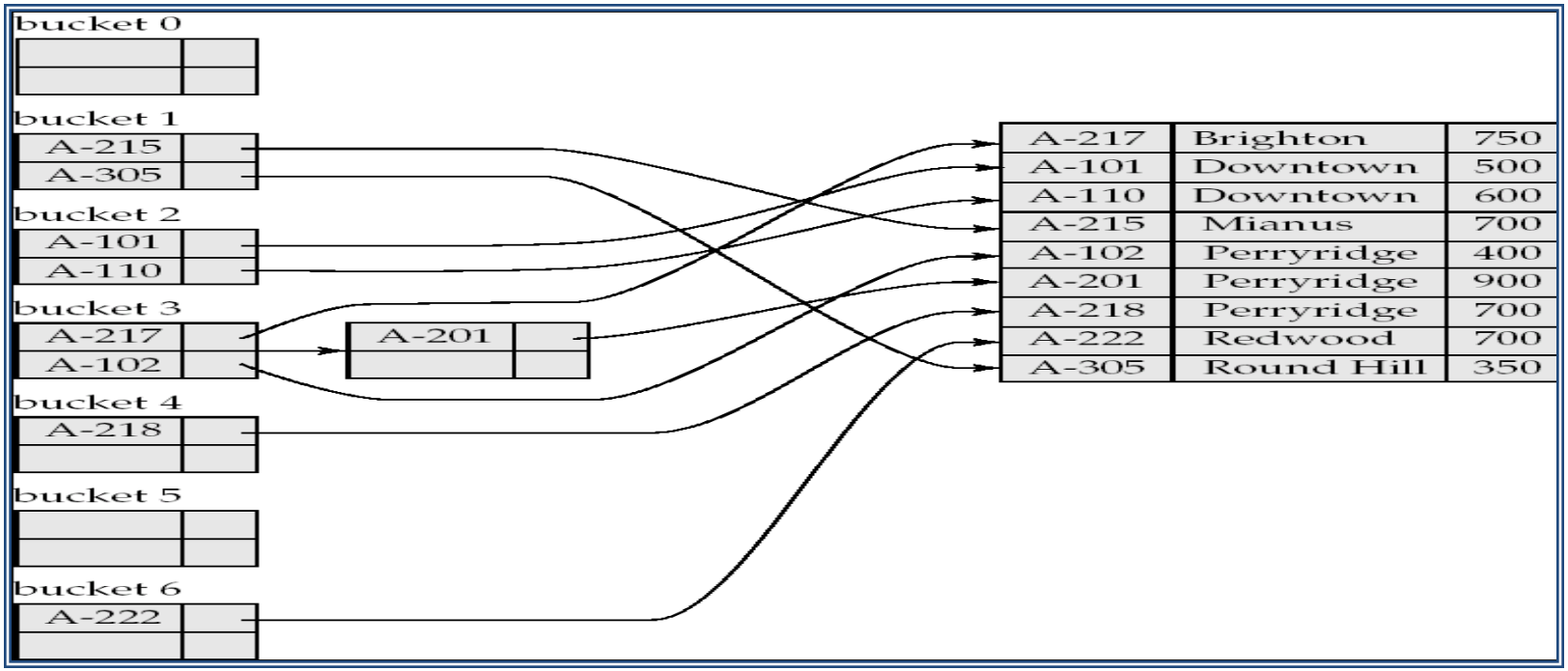
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets.*
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
  - Also called *closed hashing*
- Linear Probing - does not use overflow buckets, is not suitable for database applications.
  - Also called *open hashing*

Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A hash index organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.
- We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).

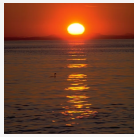
Hash Indices(contd…)

- Figure below shows a secondary hash index on the *account file, for the search key account-number. The hash function in the figure* computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2. One of the buckets has three keys mapped to it, so it has an overflow bucket



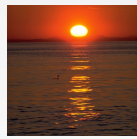
Chapter 9: Crash Recovery

Database System Concepts



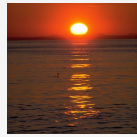
Chapter 9: Crash Recovery

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Remote Backup Systems



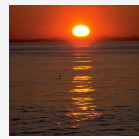
Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures



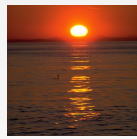
Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability



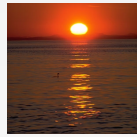
Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media



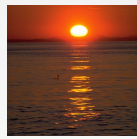
Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item *X* is called x_i.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

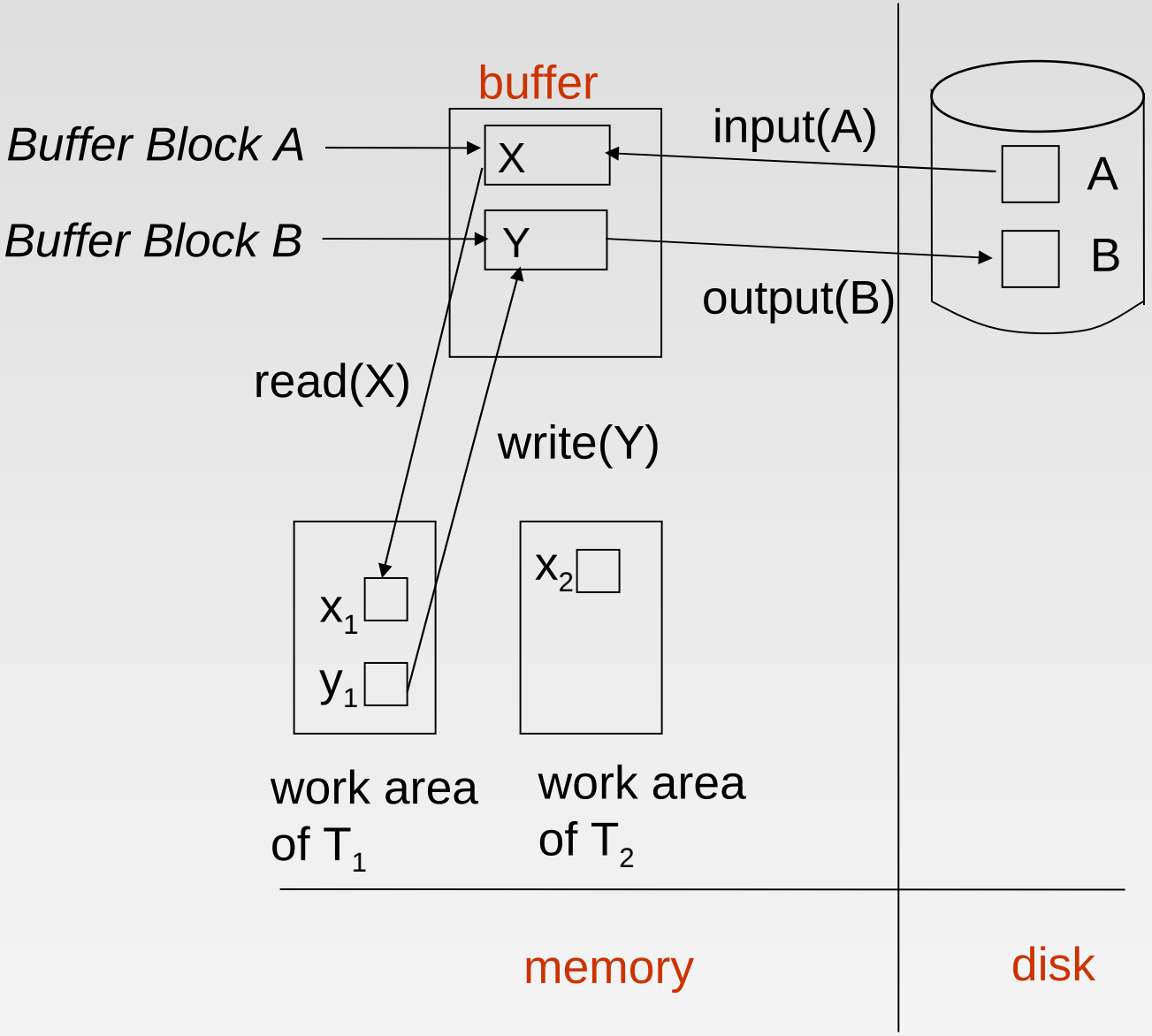


Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.
- Transactions
  - Perform **read**(X) while accessing X for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(X).
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

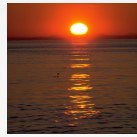


Example of Data Access



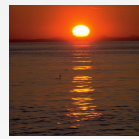
Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.



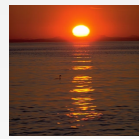
Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.



Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  $<T_i$ **start**>log record
- *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record $<T_i$ **commi**t> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification



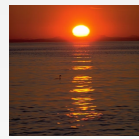
Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)	T_1 : **read** (C)
A: - A - 50	*C:- C- 100*
Write (A)	**write** (C)
read (B)	
B:- B + 50	
write (B)	



Deferred Database Modification (Cont.)

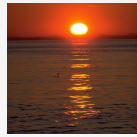
- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>

- If log on stable storage at time of crash is as in case:
  - (a) No redo actions need to be taken
  - (b) redo(T_0) must be performed since $<T_0$ **commi**t> is present
  - (c) **redo**(T_0) must be performed followed by redo(T_1) since $<T_0$ **commit**> and $<T_i$ commit> are present

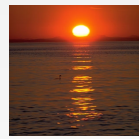
Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.



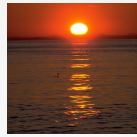
Immediate Database Modification Example

Log	Database
<T_0 **start**>	
<T_0, A, 1000, 950>	
T_0, B, 2000, 2050	
	A = 950
	B = 2050
<T_0 **commit**>	
<T_1 **start**>	
<T_1, C, 700, 600>	
	C = 600
<T_1 **commit**>	



Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.



Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

Recovery actions in each case above are:

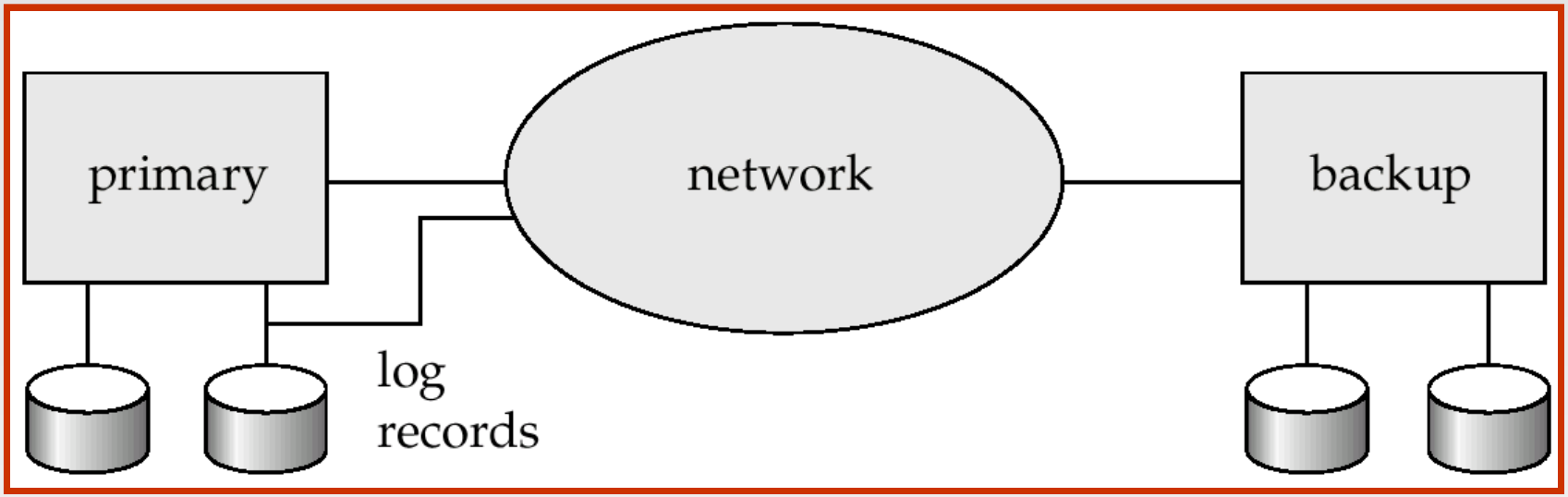
(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

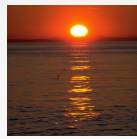
Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



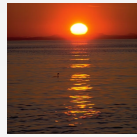
Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.



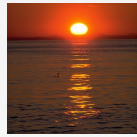
Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19



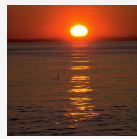
Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe:** commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

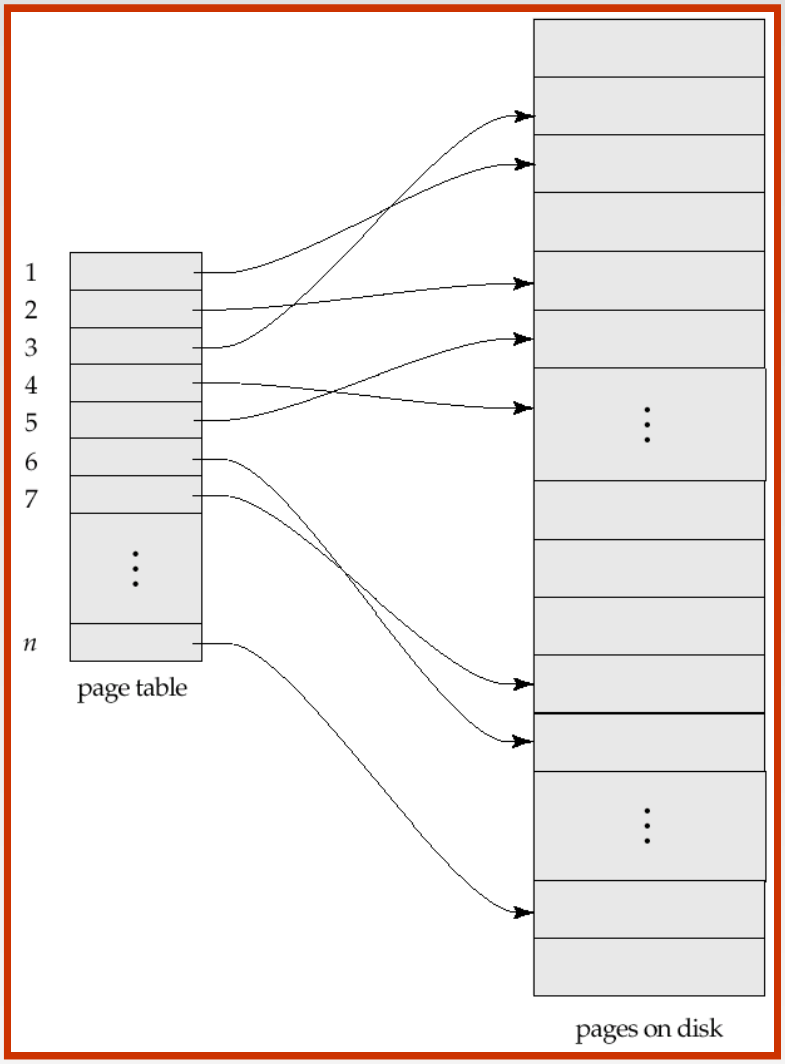


Shadow Paging

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy
  - The update is performed on the copy

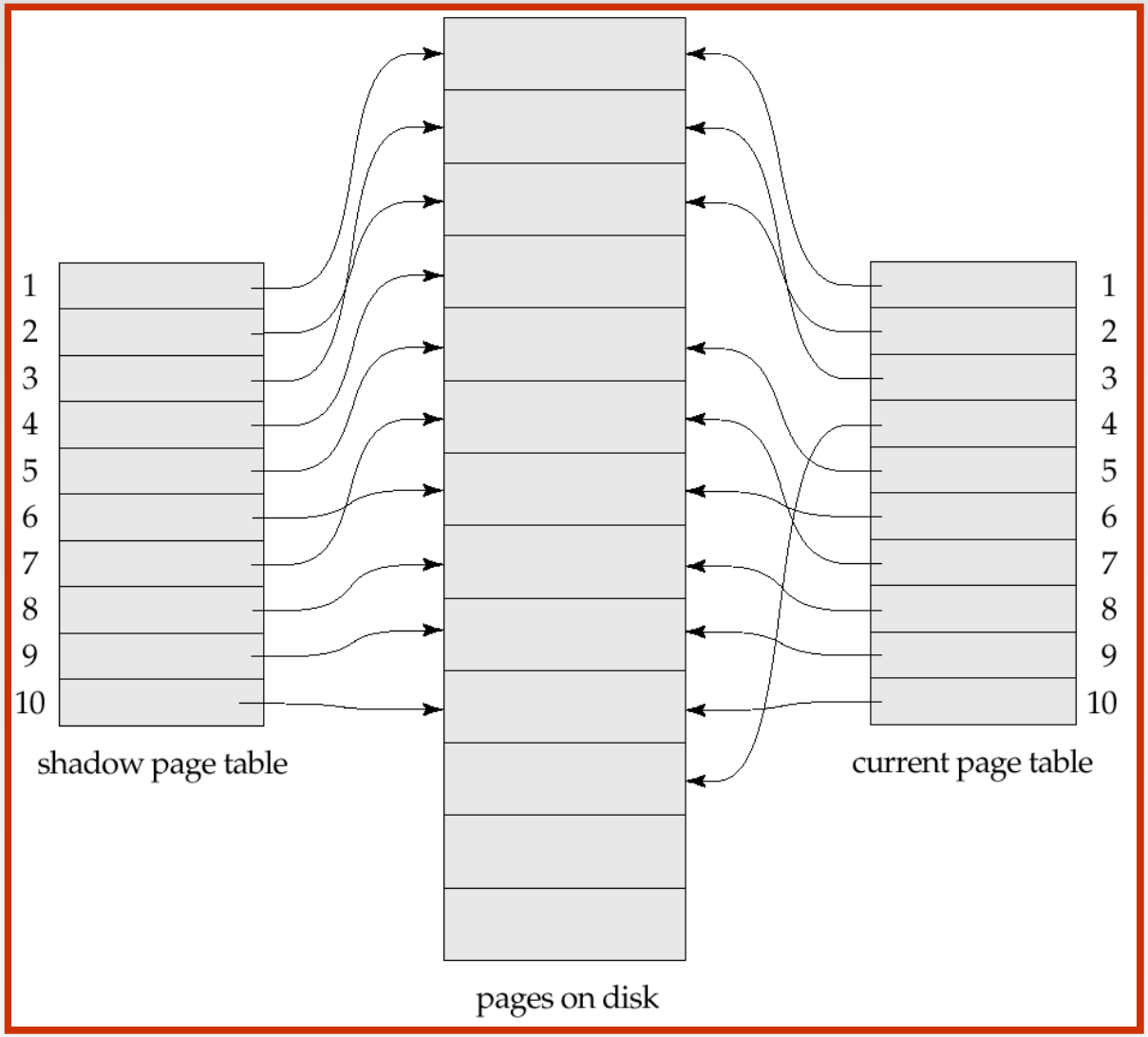


Sample Page Table



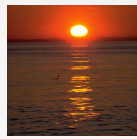
Example of Shadow Paging

Shadow and current page tables after write to page 4



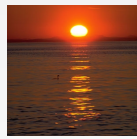
Shadow Paging (Cont.)

- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

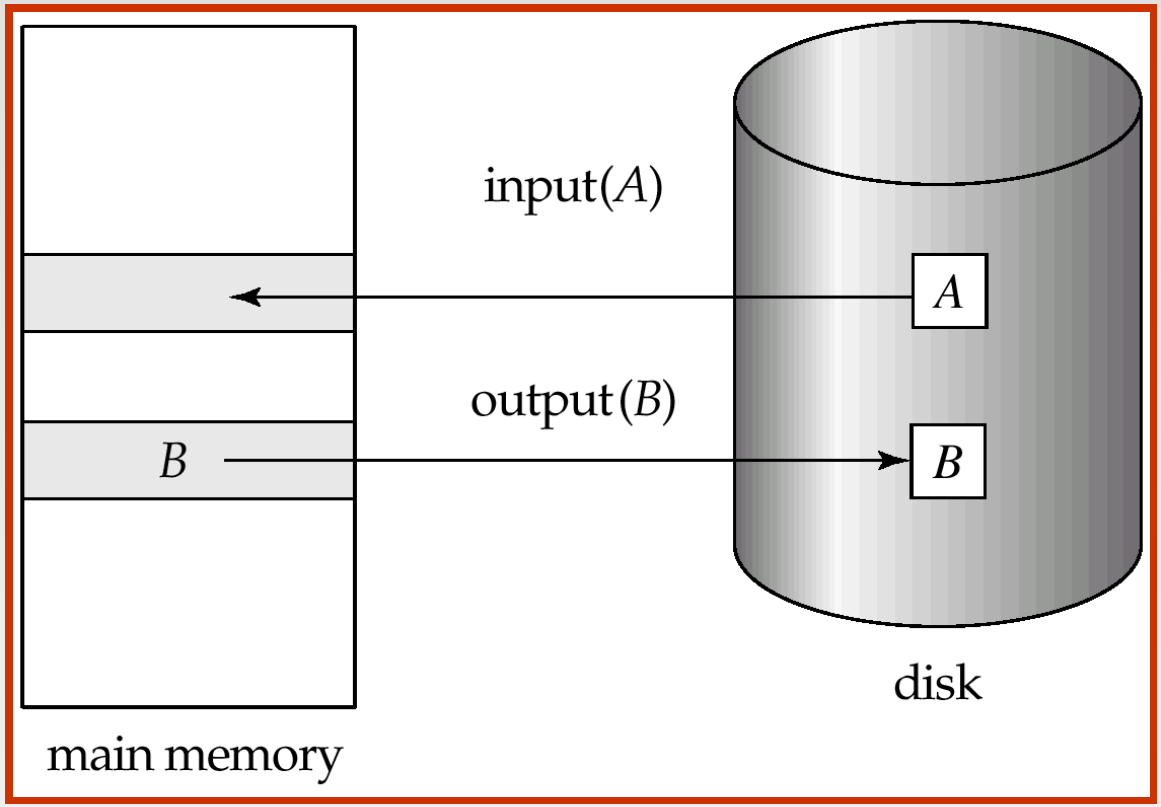


Show Paging (Cont.)

- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is trivial
- Disadvantages :
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B^+-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes



Block Storage Operations



Transaction Processing

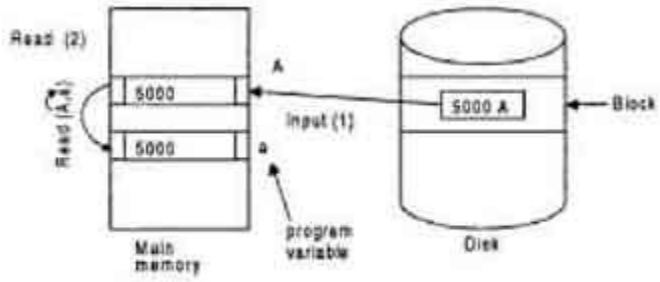
A transaction is a set of changes that must all be made together. It is a unit of program execution that access and possibly updates various data items.Moreover, it is a program unit whose execution mayor may not change the contents of a database.If the database was in consistent state before a transaction, then after execution of the transaction also, the database must be in a consistate. For example, a transfer of money from one bank account to another requires two changes to the database both must succeed or fail together.

Process of Transaction

The transaction is executed as a series of reads and writes of database objects, which are explained below:

Read Operation

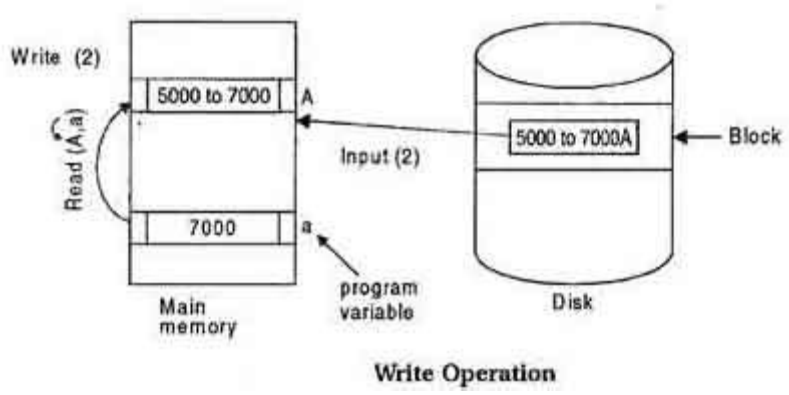
To read a database object, it is first brought into main memory from disk, and then its value is copied into a program variable as shown in figure.



Read Operation

Write Operation

To write a database object, an in-memory copy of the object is first modified and then written to disk.



Write Operation

Transaction Properties

There are four important properties of transaction that a DBMS must ensure to maintain data in the case of concurrent access and system failures. These are:

Atomicity: (all or nothing)

A transaction is said to be atomic if a transaction always executes all its actions in one step or not executes any actions at all It means either all or none of the transactions operations are performed.

Consistency: (No violation of integrity constraints)

A transaction must preserve the *consistency* of a database after the execution. The DBMS assumes that this property holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

Isolation: (concurrent changes invisibles)

The transactions must behave as if they are executed in isolation. It means that if several transactions are executed concurrently the results must be same as if they were executed serially in some order. The data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

Durability: (committed update persist)

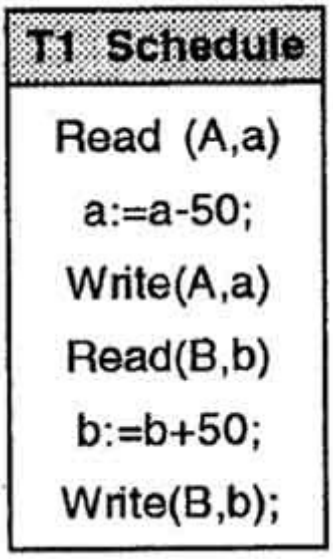
The effect of completed or committed transactions should persist even after a crash. It means once a transaction commits, the system must guarantee that the result of its operations will never be lost, in spite of subsequent failures.

The acronym ACID is sometimes used to refer above four properties of transaction that we have presented here: Atomicity, Consistency, Isolation, and Durability.

Example

In order to understand above properties consider the following example:

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



Atomicity

Suppose that, just prior to execution of transaction Ti the values of account A and B are Rs.I000 and Rs.2000.

Now, suppose that during the execution of Ti, a power failure has occurred that prevented the Ti to complete successfully. The point of failure may be after the completion Write (A,a) and before Write(B,b). It means that the changes in A are performed but not in B. Thus the values of account A and Bare Rs.950 and Rs.2000 respectively. We have lost Rs.50 as a result 'of this failure.

Now, our database is in inconsistent state.

The reason for this inconsistent state is that our transaction is completed partially and we save the changes of uncommitted transaction. So, in order to get the consistent state, database must be restored to its original values i.e. A to Rs.I000 and B to Rs.2000, this leads to the concept of atomicity of transaction. It means that in order to maintain the consistency of database, either all or none of transaction's operations are performed.

In order to maintain atomicity of transaction, the database system keeps track of the old values of any write and if the transaction does not complete its execution, the old values are restored to make it appear as the transaction never executed.

Consistency

The consistency requirement here is that the sum of A and B must be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction. It can. be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Isolation

If several transactions are executed concurrently (or in parallel), then each transaction must behave as if it was executed in isolation. It means that concurrent execution does not result an inconsistent state.For example, consider another transaction T2, which has to display the sum of account A and B. Then, its result should be Rs.3000.

Let's suppose that both Tl and T2 perform concurrently, their schedule is shown below:

T1	T2	Status
Read (A,a)		value of *A* i.e.1000 is copied to local variable a
a:=a-50		local variable *a*=950
Write(A,a)		value of local variable *a* 950 is copied to database item *A*
	Read(A,a)	value of database item *A* 950 is copied to *a*
	Read(B,b)	value of database item *B* 2000 is copied to *b*
	display(a+b)	2950 is displayed as sum of accounts *A* and *B*
Read(B,b)		value of data base item *B* 2000 is copied to local variable *b*
b:=b+50		local variable *b*=2050
Write(B,b)		value of local variable *b* 2050 is copied to database item *B*

The above schedule results inconsistency of database and it shows Rs.2950 as sum of accounts A and B instead of Rs.3000. The problem occurs because second concurrently running transaction T2, reads A and B at intermediate point and computes its sum, which results inconsistent value. Isolation property demands that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

A solution to the problem of concurrently executing transaction is to execute each transaction serially 'that is one after the other. However, concurrent execution of transaction provides significant performance benefits, so other solutions are developed they allow multiple transactions to execute concurrently.

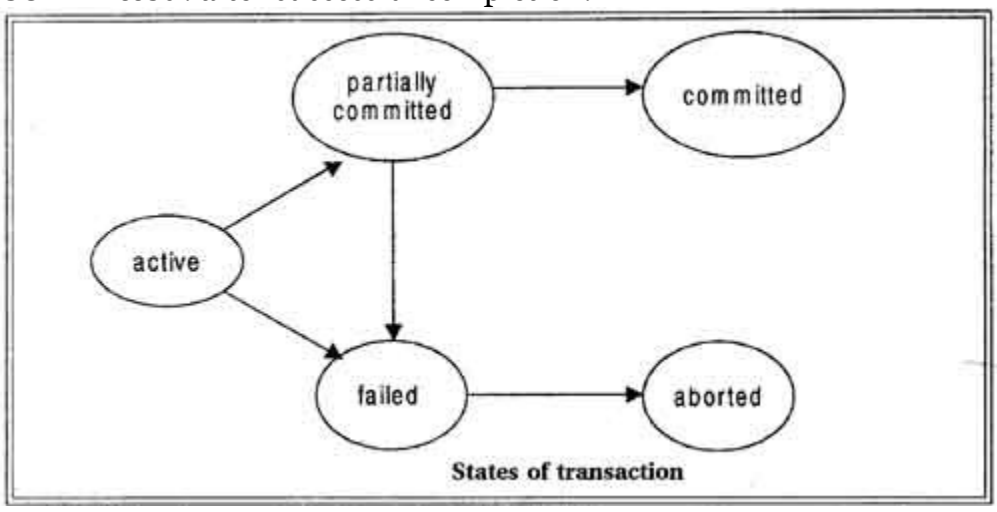
Durability

Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.

The durability property guarantees that, once a transaction completes successfully all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

States of Transaction

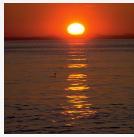
A transaction must be in one of the following states:

- **Active**: the initial state, the transaction stays in this state while it is executing.
- **Partially committed**: after the final statement has been executed.
- **Failed**: when the normal execution can no longer proceed.
- **Aborted**: after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**: after successful completion.
- 

States of transaction

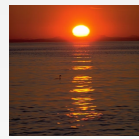
Chapter 10(contd..): Schedules and Serializability
Concepts of locking for concurrency control

Database System Concepts, 5th Ed.



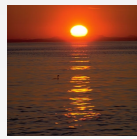
Chapter 10(contd..): Schedules and Serializability Concurrency Control

- Concurrent Executions
- Schedules
- Serializability
- Lock-Based Protocols
- Multiple Granularity



Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

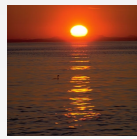
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (*A* + *B*).

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

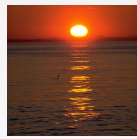


Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

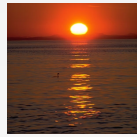
1. **conflict serializability**
2. **view serializability**

- *Simplified view of transactions*
  - We ignore operations other than **read** and **write** instructions
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  - Our simplified schedules consist of only **read** and **write** instructions.



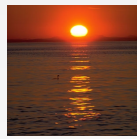
Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.



Conflict Serializability

- 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



Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

View Serializability is a process to find out that a given schedule is view serializable or not.To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule

View Equivalent:

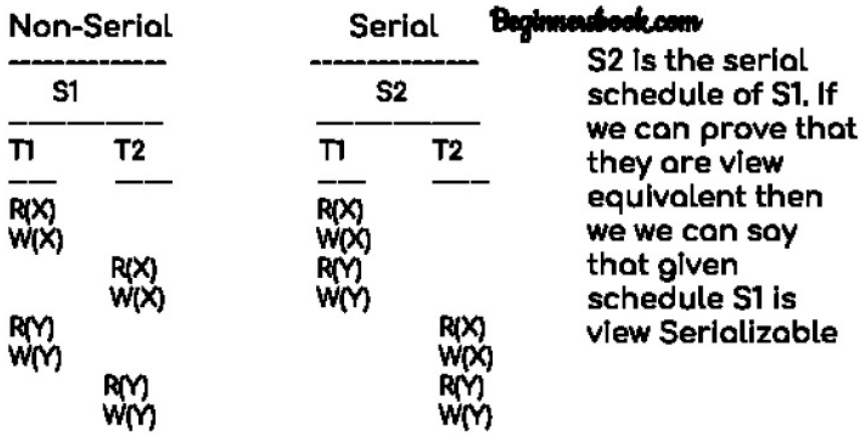
Lets learn how to check whether the two schedules are view equivalent.
Two schedules S1 and S2 are said to be view equivalent, if they satisfy all the following conditions:
1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

Read vs Initial Read: Don't confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read.

2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

View Serializable Example



Lets check the three conditions of view serializability:

Initial Read

In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

We checked for both data items X & Y and the **initial read** condition is satisfied in S1 & S2.

Final Write

In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.

Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.

We checked for both data items X & Y and the **final write** condition is satisfied in S1 & S2.

Update Read

In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.

In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.

The update read condition is also satisfied for both the schedules.

Result: Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

Note: All conflict seriable schedule are view seriable but not vice-versa.

Example:

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

Schedule S

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S1

Step 1: final updation on data items
In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.
Step 2: Initial Read
The initial read operation in S is done by T1 and in S1, it is also done by T1.
Step 3: Final Write
The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.Hence, view equivalent serial schedule is: T1 → T2 → T3

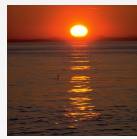
Problem : Prove whether the given schedule is View-Serializable or not?
S' : read1(A), write2(A), read3(A), write1(A), write3(A)
Solution : First of all we'll make a table for better understanding of given transactions of **schedule S'-**

T_1	T_2	T_3
read(a)		
	Write(a)	
		read(a)
Write(a)		
		write(a)

Now prove yourself

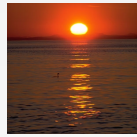
Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.



Concurrency Control vs. Serializability Tests

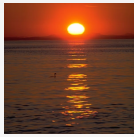
- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.



Chapter 10(contd..) : Concurrency Control

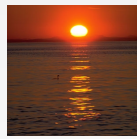
Jan 26, 2014

Database System Concepts, 5th Ed.



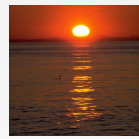
Chapter 10(contd..): Concurrency Control

- Lock-Based Protocols
- Multiple Granularity



Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.



Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- 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.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);

  read (*A*);

  unlock(*A*);

  lock-S(*B*);

  read (*B*);

  unlock(*B*);

  display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

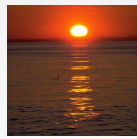
- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

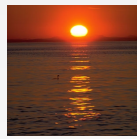
Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.



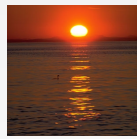
The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).



The Two-Phase Locking Protocol (Cont.)

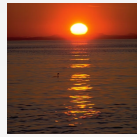
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.



The Two-Phase Locking Protocol (Cont.)

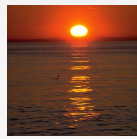
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

  Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.



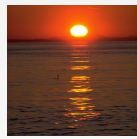
Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.



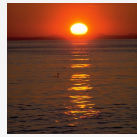
Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

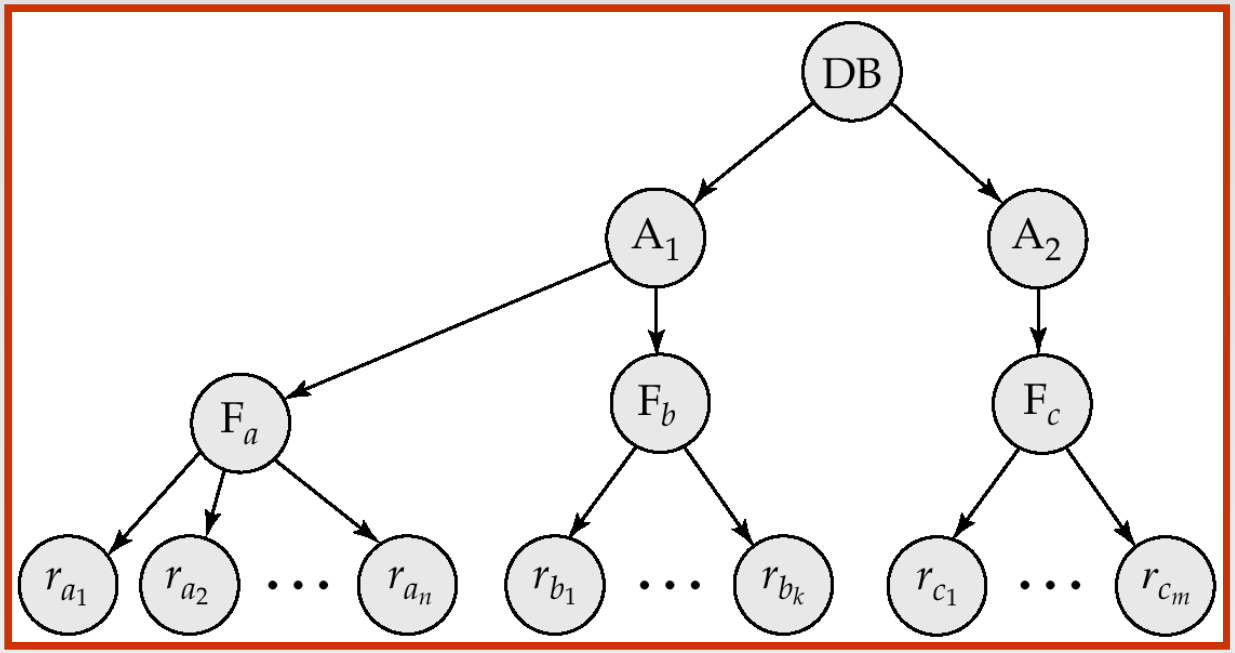


Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency



Example of Granularity Hierarchy

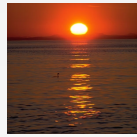


The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

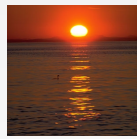
- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.



Compatibility Matrix with Intention Lock Modes

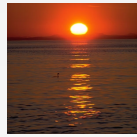
- The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×



Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  - The lock compatibility matrix must be observed.
  - The root of the tree must be locked first, and may be locked in any mode.
  - A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  - A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  - T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  - T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.



11. Advanced Database Concepts

11.1 Object-Oriented Model:

A data model is a logic organization of the real world objects (entities), constraints on them, and the relationships among objects. A DB language is a concrete syntax for a data model. A DB system implements a data model.

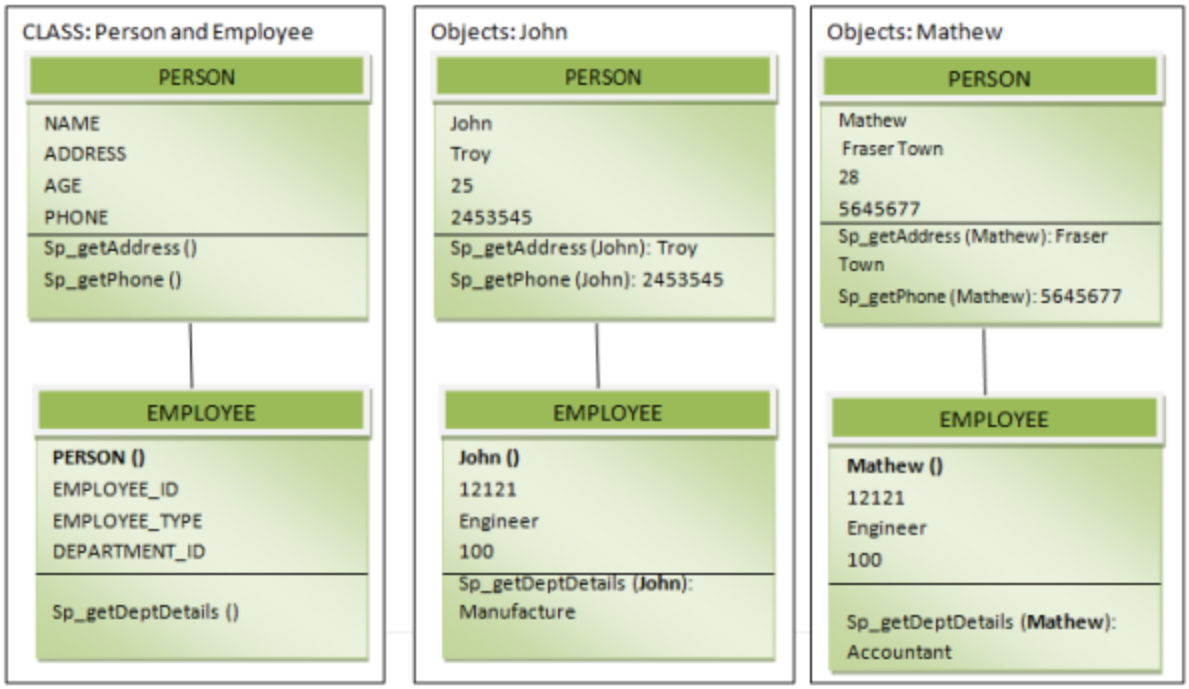
A core object-oriented data model consists of the following basic object-oriented concepts:
(1) **object and object identifier**: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).
(2) **attributes and methods**: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.
[An attribute is an instance variable, whose domain may be any class: user-defined or primitive. A class composition hierarchy (aggregation relationship) is orthogonal to the concept of a class hierarchy. The link in a class composition hierarchy may form cycles.]
(3) **class**: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.
(4) **Class hierarchy and inheritance**: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

Let us consider an Employee database to understand this model better. In this database we have different types of employees – Engineer, Accountant, Manager, Clark. But all these employees belong to Person group. Person can have different attributes like name, address, age and phone. What do we do if we want to get a person's address and phone number? We write two separate procedure sp_getAddress and sp_getPhone.

What about all the employees above? They too have all the attributes what a person has. In addition, they have their EMPLOYEE_ID, EMPLOYEE_TYPE and DEPARTMENT_ID attributes to identify them in the organization and their department. We have to retrieve their department details, and hence we sp_getDeptDetails procedure. Currently, say we need to have only these attributes and functionality.

Since all employees inherit the attributes and functionalities of Person, we can re-use those features in Employee. But do we do that? We group the features of person together into class. Hence a class has all the attributes and functionalities. For example, we would create a person class and it will have name, address, age and phone as its attribute, and sp_getAddress and sp_getPhone as procedures in it. The values for these attributes at any instance of time are object. i.e. ; {John, Troy, 25, 2453545 : sp_getAddress (John), sp_getPhone (John)} forms on person object. {Mathew, Fraser Town, 28, 5645677: sp_getAddress (Mathew), sp_getPhone (Mathew} forms another person object.

Now, we will create another class called Employee which will inherit all the functionalities of Person class. In addition it will have attributes EMPLOYEE_ID, EMPLOYEE_TYPE and DEPARTMENT_ID, and sp_getDeptDetails procedure. Different objects of Employee class are Engineer, Accountant, Manager and Clerk.



Here we can observe that the features of Person are available only if other class is inherited from it. It would be a black box to any other classes. This feature of this model is called encapsulation. It binds the features in one class and hides it from other classes. It is only visible to its objects and any inherited classes.

Advantages:

- Because of its inheritance property, we can re-use the attributes and functionalities. It reduces the cost of maintaining the same data multiple times. Also, these informations are encapsulated and, there is no fear being misused by other objects. If we need any new feature we can easily add new class inherited from parent class and adds new features. Hence it reduces the overhead and maintenance costs.
- Because of the above feature, it becomes more flexible in the case of any changes.
- Codes are re-used because of inheritance.
- Since each class binds its attributes and its functionality, it is same as representing the real world object. We can see each object as a real entity. Hence it is more understandable.

Disadvantages

- It is not widely developed and complete to use it in the database systems. Hence it is not accepted by the users.
- It is an approach for solving the requirement. It is not a technology. Hence it fails to put it in the database management systems.

11.2 Object- Relational Model (ORM)

The object-relational model is designed to provide a relational database management that allows developers to integrate databases with their data types and methods. It is essentially a relational model that allows users to integrate object-oriented features into it.
This design is most recently shown in the Nordic Object/Relational Model. The primary function of this new object-relational model is to more power, greater flexibility, better performance, and greater data integrity then those that came before it.
Some of the **benefits** that are offered by the Object-Relational Model include:

- **Extensibility** – Users are able to extend the capability of the database server; this can be done by defining new data types, as well as user-defined patterns. This allows the user to store and manage data.

- **Complex types** – It allows users to define new data types that combine one or more of the currently existing data types. Complex types aid in better flexibility in organizing the data on a structure made up of columns and tables.

- **Inheritance** – Users are able to define objects or types and tables that procure the properties of other objects, as well as add new properties that are specific to the object that has been defined.

- A field may also contain an object with attributes and operations.

- Complex objects can be stored in relational tables.

The object-relational database management systems which are also known as ORDBMS, these systems provide an addition of new and extensive object storage capabilities to the relational models at the center of the more modern information systems of today.

These services assimilate the management of conventional fielded data, more complex objects such as a time-series or more detailed geospatial data and varied dualistic media such as audio, video, images, and applets.

This can be done due to the model working to summarize methods with data structures, the ORDBMS server can implement complex analytical data and data management operations to explore and change multimedia and other more complex objects.

Disadvantages of ORDBMSs

- The ORDBMS approach has the obvious disadvantages of complexity and associated increased costs. Further, there are the proponents of the relational approach that believe the· essential simplicity' and purity of the .relational model are lost with these types of extension.

11.3 Distributed Databases:

A distributed database system consists of a collection of sites, each of which maintains a local database system. Each site is able to process local transactions: those transactions that access data in only that single site. In addition, a site may participate in the execution of global transactions; those transactions that access data in several sites. The execution of global transactions requires communication among the sites. The general structure of a distributed system appears in Figure below:

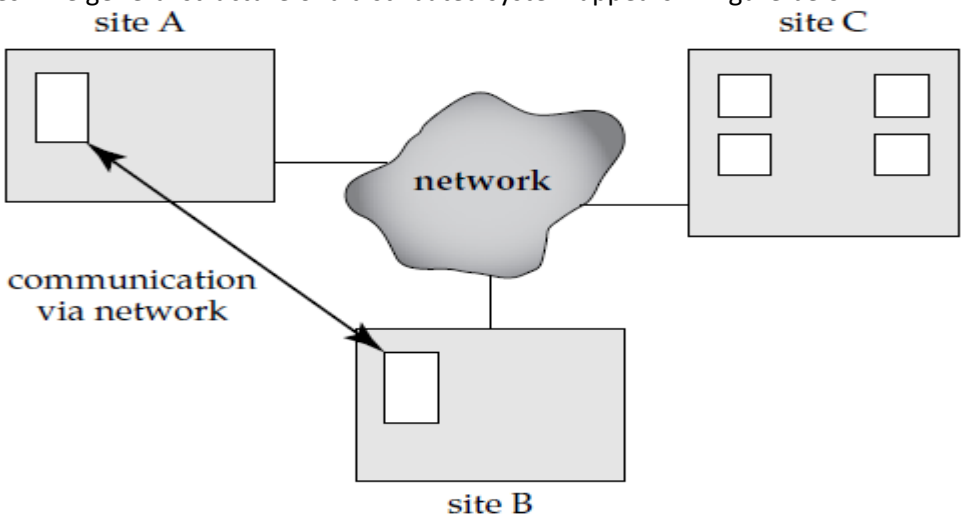


Figure 18.9 A distributed system.

There are several reasons for building distributed database systems, including sharing of data, autonomy, and availability.

• **Sharing data.** The major advantage in building a distributed database system is the provision of an environment where users at one site may be able to access the data residing at other sites. For instance, in a distributed banking system, where each branch stores data related to that branch, it is possible for a user in one branch to access data in another branch. Without this capability, a user wishing to transfer funds from one branch to another would have to resort to some external mechanism that would couple existing systems.
• **Autonomy.** The primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored locally. In a centralized system, the database administrator of the central site controls the database. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to the local database administrator for each site. Depending on the design of the distributed database system, each administrator may have a different degree of **local autonomy**. The possibility of local autonomy is often a major advantage of distributed databases.
• **Availability.** If one site fails in a distributed system, the remaining sites may be able to continue operating. In particular, if data items are **replicated** in several sites, a transaction needing a particular data item may find that item in any of several sites. Thus, the failure of a site does not necessarily imply the shutdown of the system.

Homogeneous and Heterogeneous Databases:

In a **homogeneous distributed database**, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests. In such a system, local sites surrender a portion of their autonomy in terms of their right to change schemas or database management system software. That software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites.
In contrast, in a **heterogeneous distributed database**, different sites may use different schemas, and different database management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing, while the divergence in software becomes a hindrance for processing transactions that access multiple sites.

Disadvantages of Distributed Database:

- Complexity — extra work must be done by the DBAs to ensure that the distributed nature of the system is transparent. Extra work must also be done to maintain multiple disparate systems, instead of one big one. Extra database design work must also be done to account for the disconnected nature of the database — for example, joins become prohibitively expensive when performed across multiple systems.
- Economics — increased complexity and a more extensive infrastructure means extra labour costs.
- Security — remote database fragments must be secured, and they are not centralized so the remote sites must be secured as well. The infrastructure must also be secured (e.g., by encrypting the network links between remote sites).
- Difficult to maintain integrity — but in a distributed database, enforcing integrity over a network may require too much of the network's resources to be feasible.
- Additional software is required.
- Concurrency control: it is a major issue. It can be solved by locking and timestamping.

11.4 Concepts of Data Warehouses:

A **data warehouse** is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write. Moreover, by accessing information for decision support from a data warehouse, the decision maker ensures that online transaction-Processing systems are not affected by the decision-support workload.

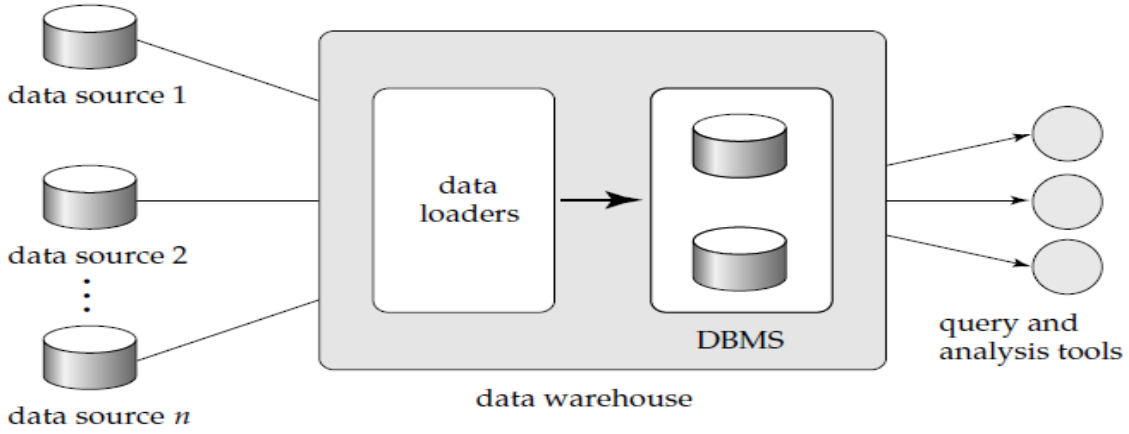


Figure 22.8 Data-warehouse architecture.

Components of a Data Warehouse:

Figure 22.8 shows the architecture of a typical data warehouse, and illustrates the gathering of data, the storage of data, and the querying and data-analysis support. Among the issues to be addressed in building a warehouse are the following:

• **When and how to gather data.** In a **source-driven architecture** for gathering data, the data sources transmit new information, either continually (as transaction processing takes place), or periodically (nightly, for example). In a **destination-driven architecture**, the data warehouse periodically sends requests for new data to the sources.

• **What schema to use.** Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration, and to convert data to the integrated schema before they are stored. As a result, the data stored in the warehouse are not just a copy of the data at the sources. Instead, they can be thought of as a materialized view of the data at the sources.

• **Data cleansing.** The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, that can be corrected. For example, names are often misspelled, and addresses may have street/area/city names misspelled, or zip codes entered incorrectly. These can be corrected to a reasonable extent by consulting a database of street names and zip codes in each city. Address lists collected from multiple sources may have duplicates that need to be eliminated in a **merge– purge operation**. Records for multiple individuals in a house may be grouped together so only one mailing is sent to each house; this operation is called **householding**.

• **How to propagate updates**. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward. If they are not, the problem of propagating updates is basically the *view-maintenance* problem.

• **What data to summarize.** The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation. For example, instead of storing data about every sale of clothing, we can store total sales of clothing by item name and category.