1

OVERVIEW

Topics Covered:

1.1 Database management system
1.2 Data Independence
1.3 Data Abstraction
1.4 Data Models
1.5 DBMS Architecture
1.6 Users of DBMS
1.7 Overview of Conventional Data Models

1.1 DATABASE MANAGEMENT SYSTEM (DBMS) DEFINITION:-

A database management system is a collection of interrelated data and a set of programs to access those data. Collection of data is referred to as a database.

Primary goal of dbms is to provide a way to store and retrieve database information that is both convenient and efficient.

Dbms allows us to define structure for storage of information and also provides mechanism to manipulate this information. Dbms also provides safety for the information stored despite system crashes or attempts of authorized access.

Limitations of data processing environment:-

1) Data redundancy and consistency:- Different files have different formats of programs written in different programming languages by different users. So the same information may be duplicated in several files. It may lead to data inconsistency.

   If a customer changes his address, then it may be reflected in one copy of data but not in the other.

2) Difficulty in accessing data:- The file system environment does not allow needed data to be retrieved in a convenient and efficient manner.

3) Data isolation:- Data is scattered in various files; so it gets isolated because file may be in different formats.

4) Integrity problems:- Data values stored in the database must satisfy consistency constraints. Problem occurs when constraints involve several data items from different files.

5) Atomicity problems:- If failure occurs, data must be stored to constant state that existed prior to failure. For example, if in a bank account, a person abc is transferring Rs 5000 to the account of pqr, and abc has withdrawn the money but before it gets deposited to the pqr's account, the system failure occurs, then Rs5000 should be deposited back to abc's bank account.

6) Concurrent access anomalies:- Many systems allow multiple users to update data simultaneously. Concurrent updates should not result in inconsistent data.

7) Security problems:- Not every user of the database system should be able to access all data. Data base should be protected from access by unauthorized users.

1.2 DATA INDEPENDENCE

We can define two types of data independence:

1. **Logical data independence**:

It is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence**:

It is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by

referring to the mapping information in the catalog. Data independence is accomplished because, when the schema is changed at some level, the schema at the next higher level remains unchanged; only the mapping between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

1.3 DATA ABSTRACTION:

Major purpose of dbms is to provide users with abstract view of data i.e. the system hides certain details of how the data are stored and maintained.

Since database system users are not computer trained, developers hide the complexity from users through *3 levels of abstraction*, to simplify user's interaction with the system.

1) **Physical level of data abstraction:**
   This s the lowest level of abstraction which describes how data are actually stored.

2) **Logical level of data abstraction:**
   This level hides what data are actually stored in the database and what relationship exists among them.

3) **View Level of data abstraction:**
   View provides security mechanism to prevent user from accessing certain parts of database.

1.4 DATA MODELS

Many data models have been proposed, and we can categorize them according to the types of concepts they use to describe the database structure.

High-level or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users. Between these two extremes is a class of **representational (or implementation) data models**, which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Representational data models hide some details of data storage but can be implemented on a computer system in a direct way. Conceptual data models use concepts such as entities, attributes, and relationships.

An entity represents a real-world object or concept, such as an employee or a project, that is described in the database. An attribute represents some property of interest that further describes an entity, such as the employee's name or salary. A relationship among two or more entities represents an interaction among the entities, which is explained by the **Entity-Relationship model**—a popular high-level conceptual data model.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used relational data model, as well as the so-called legacy data models—the network and hierarchical models—that have been widely used in the past.

We can regard object data models as a new family of higher-level implementation data models that are closer to conceptual data models.

Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored in the computer by representing information such as record formats, record orderings, and access paths. An access path is a structure that makes the search for particular database records efficient.

1.5 DBMS ARCHITECTURE

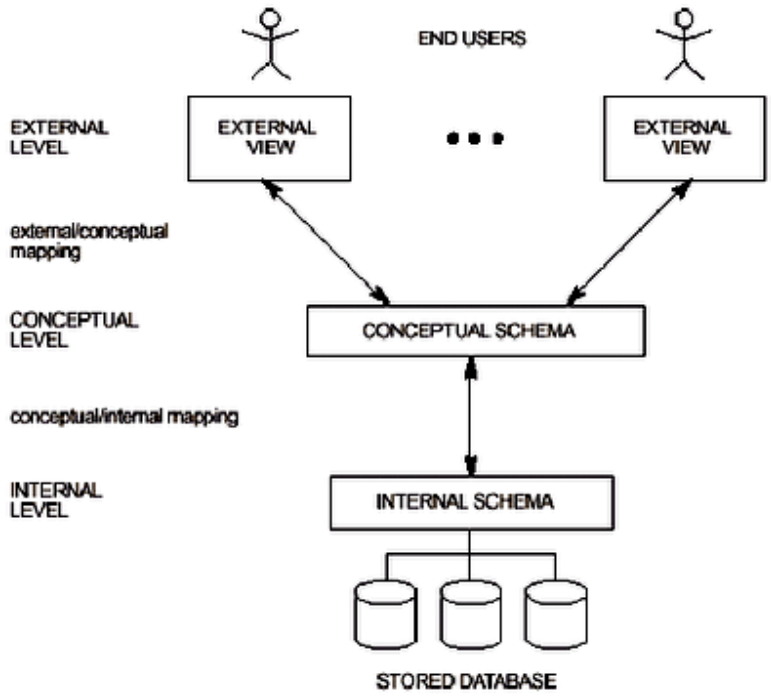


Fig: Three-Schema DBMS Architecture

The goal of the three-schema architecture, illustrated in above Figure, is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The internal level has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

2. The conceptual level has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

3. **The external** or **view level** includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels. Notice that the three schemas are only descriptions of data; the only data that actually exists is at the physical level. In a DBMS based on the three-schema architecture, each user group refers only to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called mappings. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

1.6 PEOPLE WHO WORK WITH THE DATABASE:

The people who use the database can be categorized
a) **Database users**
b) **Database administrator (DBA).**

a) Database users are of 4 different types:

1) Naive users:
These are the unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.

E.g. consider a user who checks for account balance information over the World Wide Web. Such a user access a form, enters the account number and password etc. And the application program on the internet then retrieves the account balance using given account information which s passed to the user.

2) Application programmers:
These are computer professionals who write application programs, used to develop user interfaces. The application programmer uses Rapid Application Development (RAD) toolkit or special type of programming languages which include special features to facilitate generation of forms and display of date on screen.

3) Sophisticated users:
These users interact with the database using database query language. They submit their query to the query processor. Then Data Manipulation Language (DML) functions are performed on the database to retrieve the data. Tools used by these users are OLAP(Online Analytical Processing) and data mining tools.

4) Specialized users:
These users write specialized database applications to retrieve data. These applications can be used to retrieve data with complex data types e.g. graphics data and audio data.

b) Database Administrator (DBA)
A person having who has central control over data and programs that access the data is called DBA. Following are the functions of the DBA.

1) Schema definition: DBA creates database schema by executing Data Definition Language (DDL) statements.

2) Storage structure and access method definition

3) Schema and physical organization modification: If any changes are to be made in the original schema, to fit the need of your organization, then these changes are carried out by the DBA.

4) Granting of authorization foe data access: DBA can decide which parts of data can be accessed by which users. Before any user access the data, dbms checks which rights are granted to the user by the DBA.

5) Routine maintenance: DBA has to take periodic backups of the database, ensure that enough disk space is available to store new data, ensure that performance of dbms ix not degraded by any operation carried out by the users.

1.7 OVERVIEW OF CONVENTIONAL DATA MODELS:

1.7.1 Hierarchical Data Model:

One of the most important applications for the earliest database management systems was production planning for manufacturing companies. If an automobile manufacturer decided to produce 10,000 units of one car model and 5,000 units of another model, it needed to know how many parts to order from its suppliers. To answer the question, the product (a car) had to be decomposed into assemblies (engine, body, chassis), which were decomposed into subassemblies (valves, cylinders, spark plugs), and then into sub-subassemblies, and so on. Handling this list of parts, known as a bill of materials, was a job tailor-made for computers. The bill of materials for a product has a natural hierarchical structure. To store this data, the hierarchical data model, illustrated in Figure below was developed. In this model, each record in the database represented a specific part. The records had parent/child relationships, linking each part to its subpart, and so on.

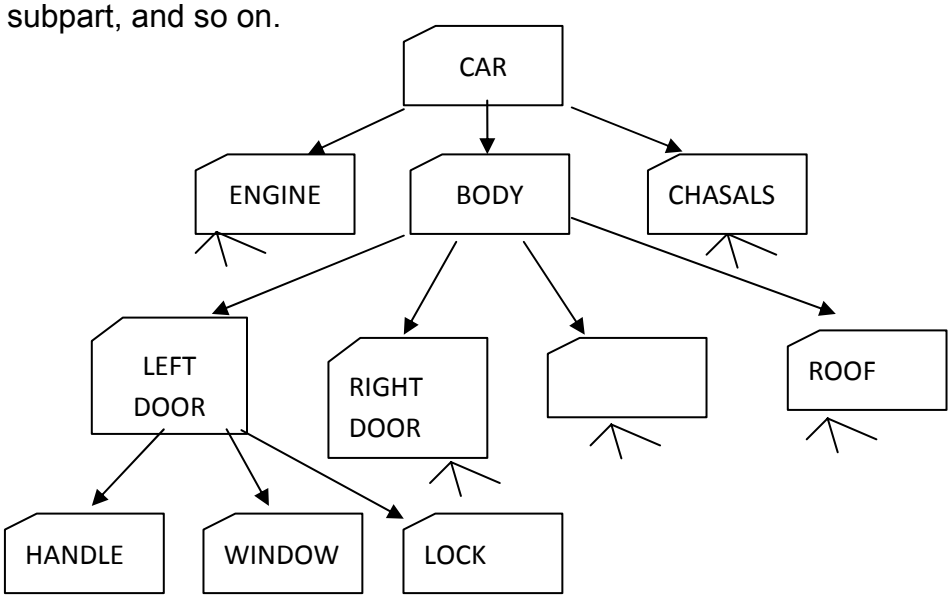


Figure 4-2: A hierarchical bill-of-materials database

To access the data in the database, a program could:

- find a particular part by number (such as the left door),
- move "down" to the first child (the door handle),
- move "up" to its parent (the body), or
- move "sideways" to the next child (the right door).

Retrieving the data in a hierarchical database thus required navigating through the records, moving up, down, and sideways one record at a time.

One of the most popular hierarchical database management systems was IBM's
Information Management System (IMS), first introduced in 1968.

The advantages of IMS and its hierarchical model are as follows:

• **Simple structure**: The organization of an IMS database was easy to understand. The database hierarchy paralleled that of a company organization chart or a family tree.

• **Parent/child organization**: An IMS database was excellent for representing parent/child relationships, such as "A is a part of B" or "A is owned by B."

• **Performance**: IMS stored parent/child relationships as physical pointers from one data record to another, so that movement through the database was rapid. Because the structure was simple, IMS could place parent and child records close to one another on the disk, minimizing disk input/output.

IMS is still a very widely used DBMS on IBM mainframes. Its raw performance makes it the database of choice in high-volume transaction processing applications such as processing bank ATM transactions, verifying credit card numbers, and tracking the delivery of overnight packages. Although relational database performance has improved dramatically over the last decade, the performance requirements of applications such as these have also increased, insuring a continued role for IMS.

1.7.2 Network Data Model:

The simple structure of a hierarchical database became a disadvantage when the data had a more complex structure. In an order-processing database, for example, a single order might participate in three different parent/child relationships, linking the order to the customer who placed it, the salesperson who took it, and the product ordered. The structure of this type of data simply didn't fit the strict hierarchy of IMS.

To deal with applications such as order processing, a new network data model was developed. The network data model extended the hierarchical model by allowing a record to participate in multiple parent/child relationships.

For a programmer, accessing a network database was very similar to accessing a hierarchical database. An application program could:

• find a specific parent record by key (such as a customer number),

• move down to the first child in a particular set (the first order placed by this customer),

• move sideways from one child to the next in the set (the next order placed by the same customer), or

• move up from a child to its parent in another set (the salesperson who took the order).

Once again the programmer had to navigate the database record-by-record, this time specifying which relationship to navigate as well as the direction.

Network databases had several advantages:

• **Flexibility**: Multiple parent/child relationships allowed a network database to represent data that did not have a simple hierarchical structure.

• **Standardization**: The CODASYL standard boosted the popularity of the network model, and minicomputer vendors such as Digital Equipment Corporation and Data General implemented network databases.

• **Performance:** Despite their greater complexity, network databases boasted performance approaching that of hierarchical databases. Sets were represented by pointers to physical data records, and on some systems, the database administrator could specify data clustering based on a set relationship.

Network databases had their disadvantages, too. Like hierarchical databases, they were very rigid. The set relationships and the structure of the records had to be specified in advance. Changing the database structure typically required rebuilding the entire database.

2

ENTITY RELATIONSHIP MODEL

Topics Covered:

2.1 Entity
2.2 Attributes
2.3 Keys
2.4 Relation
2.5 Cardinality
2.6 Participation
2.7 Weak Entities
2.8 ER Diagram
2.9 Conceptual Design With ER Model

2.1 ENTITY

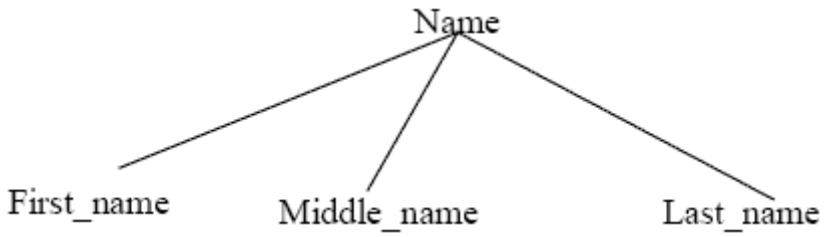
- The basic object that the ER model represents is an **entity**, which is a "thing" in the real world with an independent existence.
- An entity may be an object with a physical existence—a particular person, car, house, or employee—or it may be an object with a conceptual existence—a company, a job, or a university course.

2.2 ATTRIBUTES

- 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.
- Several types of attributes occur in the ER model: *simple versus composite*; *single-valued versus multi-valued*; and *stored versus derived*.

2.2.1 Composite versus Simple (Atomic) Attributes

- Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings.
- For example, the Address attribute of the employee entity can be sub-divided into Street_Name, City, State, and Zip.
- Attributes that are not divisible are called *simple* or *atomic* attributes.
- Composite attributes can form a hierarchy; for example, Name can be subdivided into three simple attributes, First_Name, Middle Name, Last_Name.
- The value of a composite attribute is the concatenation of the values of its constituent simple attributes.



Composite Attributes

2.2.2 Single-valued Versus Multi-valued Attributes

- Attributes which have only one value for a entity are called single valued attributes.
- E.g. For a student entity, RollNo attribute has only one single value.
- But phone number attribute may have multiple values. Such values are called Multi-valued attributes.

2.2.3 Stored Versus Derived Attributes

- Two or more attribute values are related—for example, the Age and Birth Date attributes of a person.
- For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth Date.
- The Age attribute is hence called a *derived*
- The attribute from which another attribute value is derived is called *stored attribute*.
- In the above example, date of birth is the stored attribute.

- Take another example, if we have to calculate the interest on some principal amount for a given time, and for a particular rate of interest, we can simply use the interest formule
    - *Interest=NPR/100;*
- In this case, interest is the derived attribute whereas principal amount(P), time(N) and rate of interest(R) are all stored attributes.

2.3 KEYS

- An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes.
- A key is an attribute (also known as column or field) or a combination of attribute that is used to identify records.
- Sometimes we might have to retrieve data from more than one table, in those cases we require to join tables with the help of keys.
- The purpose of the key is to bind data together across tables without repeating all of the data in every table
- Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely.
- For example, the Name attribute is a key of the COMPANY entity type because no two companies are allowed to have the same name.
- For the PERSON entity type, a typical key attribute is SocialSecurityNumber.
- Sometimes, several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity.
- If a set of attributes possesses this property, we can define a composite attribute that becomes a key attribute of the entity type.

The various types of key with e.g. in SQL are mentioned below, (For examples let suppose we have an Employee Table with attributes ‘ID’ , ‘Name’ ,’Address’ , ‘Department_ID’ ,’Salary’)

(I) Super Key – An attribute or a combination of attribute that is used to identify the records uniquely is known as Super Key. A table can have many Super Keys.

E.g. of Super Key

1 ID
2 ID, Name
3 ID, Address
4 ID, Department_ID
5 ID, Salary
6 Name, Address
7 Name, Address, Department_ID …… So on as any combination which can identify the records uniquely will be a Super Key.

(II) Candidate Key – It can be defined as minimal Super Key or irreducible Super Key. In other words an attribute or a combination of attribute that identifies the record uniquely but none of its proper subsets can identify the records uniquely.

E.g. of Candidate Key

1 Code
2 Name, Address

For above table we have only two Candidate Keys (i.e. Irreducible Super Key) used to identify the records from the table uniquely. Code Key can identify the record uniquely and similarly combination of Name and Address can identify the record uniquely, but neither Name nor Address can be used to identify the records uniquely as it might be possible that we have two employees with similar name or two employees from the same house.

(III) Primary Key – A Candidate Key that is used by the database designer for unique identification of each row in a table is known as Primary Key. A Primary Key can consist of one or more attributes of a table.

E.g. of Primary Key - Database designer can use one of the Candidate Key as a Primary Key. In this case we have “Code” and “Name, Address” as Candidate Key, we will consider “Code” Key as a Primary Key as the other key is the combination of more than one attribute.

(IV) Foreign Key – A foreign key is an attribute or combination of attribute in one base table that points to the candidate key (generally it is the primary key) of another table. The purpose of the foreign key is to ensure referential integrity of the data i.e. only values that are supposed to appear in the database are permitted.

E.g. of Foreign Key – Let consider we have another table i.e. Department Table with Attributes "Department_ID", "Department_Name", "Manager_ID", "Location_ID" with Department_ID as an Primary Key. Now the Department_ID attribute of Employee Table (dependent or child table) can be defined as the Foreign Key as it can reference to the Department_ID attribute of the Departments table (the referenced or parent table), a Foreign Key value must match an existing value in the parent table or be NULL.

(V) Composite Key – If we use multiple attributes to create a Primary Key then that Primary Key is called Composite Key (also called a Compound Key or Concatenated Key).

E.g. of Composite Key, if we have used "Name, Address" as a Primary Key then it will be our Composite Key.

(VI) Alternate Key – Alternate Key can be any of the Candidate Keys except for the Primary Key.

E.g. of Alternate Key is "Name, Address" as it is the only other Candidate Key which is not a Primary Key.

(VII) Secondary Key – The attributes that are not even the Super Key but can be still used for identification of records (not unique) are known as Secondary Key.

E.g. of Secondary Key can be Name, Address, Salary, Department_ID etc. as they can identify the records but they might not be unique.

2.4 RELATION

- There are several implicit relationships among the various entity types.
- In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists.
- For example, the attribute Manager of department refers to an employee who manages the department.
- In the ER model, these references should not be represented as **relationships** or **relation**. There is a relation "borrower" in the entities customer and account which can be shown as follows:

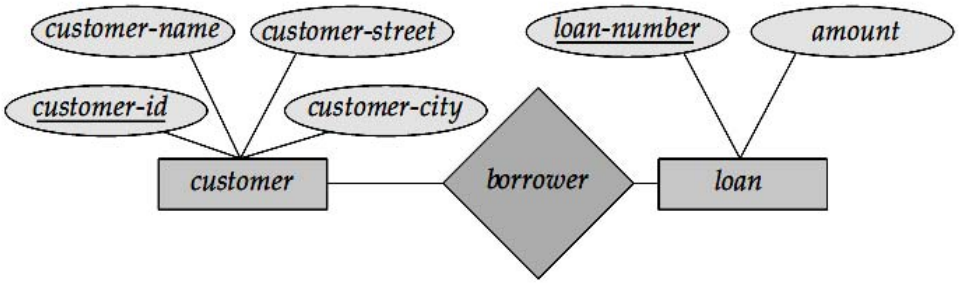


Figure: E-R diagram corresponding to customers and loans.

2.5 CARDINALITY

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

For a relationship set R between entity sets A and B, the mapping cardinality must be one of the following:
There are three types of relationships
1) One to one
2) One to many
3) Many to many

2.5.1 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

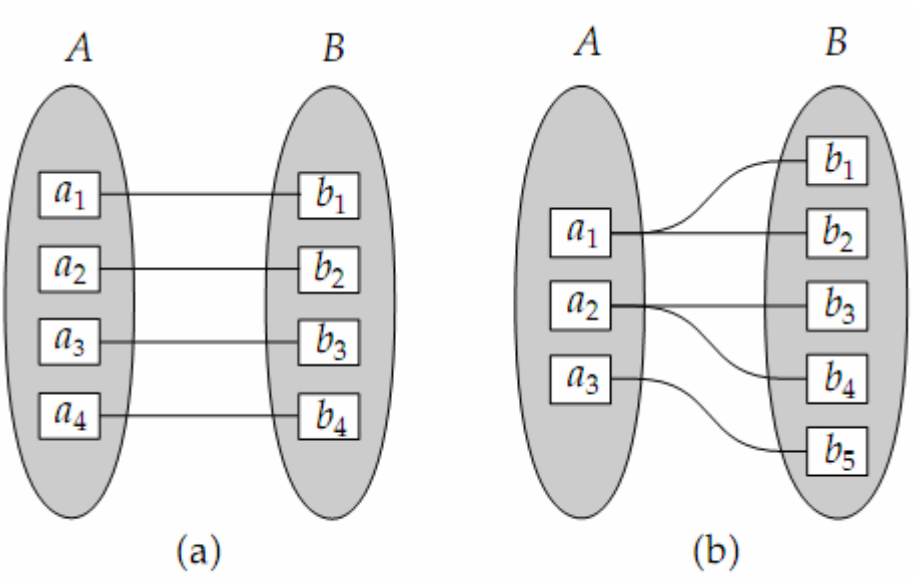
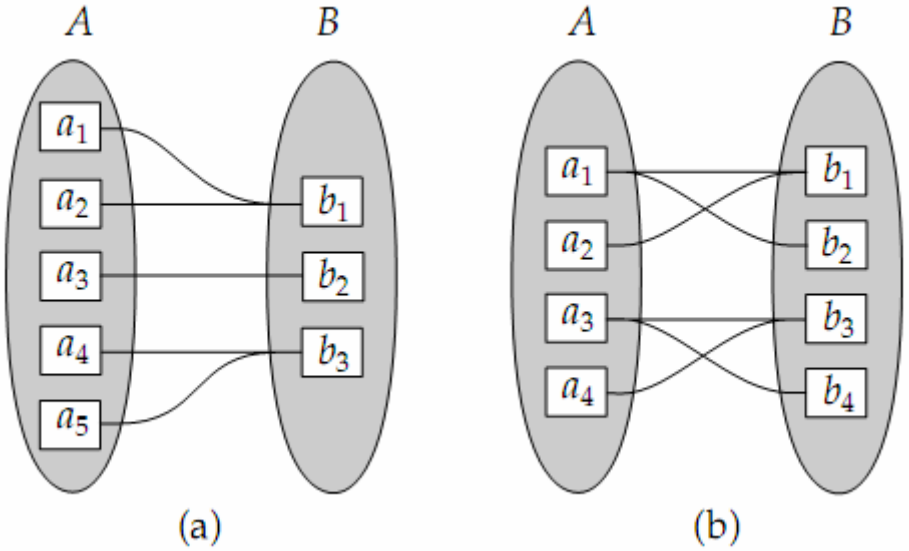


Figure: Mapping cardinalities. (a) One to one. (b) One to many.



Figure; Mapping cardinalities. (a) Many to one. (b) Many to many

2.6 PARTICIPATION

- 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.

2.7 WEAK ENTITIES

- An entity set may not have sufficient attributes to form a primary key.
- Such an entity set is termed a *weak entity set*.
- An entity set that has a primary key is termed a strong entity set.
- As an illustration, consider the entity set payment, which has the three attributes: payment-number, payment-date, and payment-amount.
- Payment numbers are typically sequential numbers, starting from 1, generated separately for each loan.
- Thus, al-though each payment entity is distinct, payments for different loans may share the same payment number. Thus, this entity set does not have a primary key; it is a weak entity set.
- For a weak entity set to be meaningful, it must be associated with another entity set, called the identifying or owner entity set.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be existence dependent on the identifying entity set.
- The identifying entity set is said to own the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the identifying relationship.
- The identifying relationship is many to one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.
- In our example, the identifying entity set for payment is loan, and a relationship loan-payment that associates payment entities with their corresponding loan entities is the identifying relationship.
- Although a weak entity set does not have a primary key, we nevertheless need a means of distinguishing among all those

entities in the weak entity set that depend on one particular strong entity.

- The discriminator of a weak entity set is a set of attributes that allows this distinction to be made.
- For example, the discriminator of the weak entity set payment is the attribute payment-number, since, for each loan, a payment number uniquely identifies one single payment for that loan.
- The discriminator of a weak entity set is also called the partial key of the entity set.
- The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.
- In the case of the entity set payment, its primary key is {loan-number, payment-number}, where loan-number is the primary key of the identifying entity set, namely loan, and payment-number distinguishes payment entities within the same loan.
- The identifying relationship set should have no descriptive attributes, since any required attributes can be associated with the weak entity set
- A weak entity set can participate in relationships other than the identifying relationship.
- For instance, the payment entity could participate in a relationship with the account entity set, identifying the account from which the payment was made.
- A weak entity set may participate as owner in an identifying relationship with another weak entity set.
- It is also possible to have a weak entity set with more than one identifying entity set.
- A particular weak entity would then be identified by a combination of entities, one from each identifying entity set.
- The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set.
- In E-R diagrams, a doubly outlined box indicates a weak entity set, and a doubly outlined diamond indicates the corresponding identifying relationship.
- The weak entity set payment depends on the strong entity set loan via the relationship set loan-payment.
- The figure also illustrates the use of double lines to indicate total participation—the participation of the (weak) entity set payment in the relationship loan-payment is total, meaning that every payment must be related via loan-payment to some loan.

Finally, the arrow from loan-payment to loan indicates that each payment is for a single loan. The discriminator of a weak entity set also is underlined, but with a dashed, rather than a solid, line.

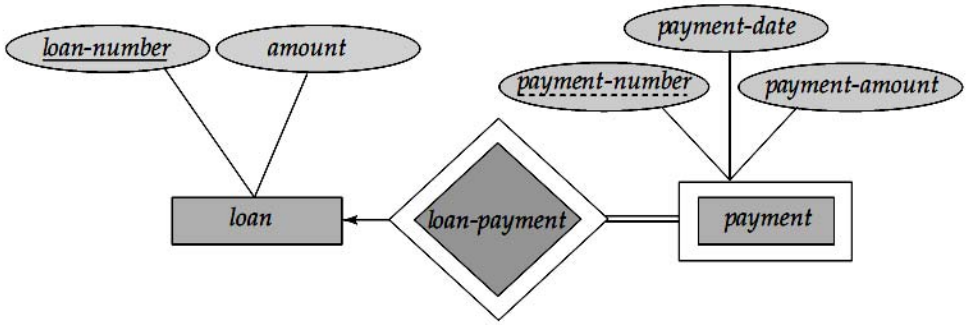


Figure: E-R diagram with a weak entity set.

2.8 ER DIAGRAM- SPECIALIZATION, GENERALIZATION AND AG GREGATION

2.8.1 Specialization:

- An entity set may include sub groupings 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 sub groupings 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.

- As another example, suppose the bank wishes to divide accounts into two categories, checking account and savings account. Savings accounts need a minimum balance, but the bank may set interest rates differently for different customers, offering better rates to favored customers.
- Checking accounts have a fixed interest rate, but offer an overdraft facility; the overdraft amount on a checking account must be recorded.
- The bank could then create two specializations of account, namely savings-account and checking-account.
- As we saw earlier, account entities are described by the attributes account-number and balance.
- The entity set savings-account would have all the attributes of account and an additional attribute interest-rate.
- The entity set checking-account would have all the attributes of account, and an additional attribute overdraft-amount.
- We can apply specialization repeatedly to refine a design scheme. For instance, bank employees may be further classified as one of the following:
    - Officer
    - Teller
    - Secretary
- Each of these employee types is described by a set of attributes that includes all the attributes of entity set employee plus additional attributes. For example, officer entities may be described further by the attribute office-number, teller entities by the attributes station-number and hours-per-week, and secretary entities by the attribute hours-per-week. Further, secretary entities may participate in a relationship secretary-for, which identifies which employees are assisted by a secretary.
- An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs. Another, coexistent, specialization could be based on whether the person is a temporary (limited-term) employee or a permanent employee, resulting in the entity sets temporary-employee and permanent-employee. When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations. For instance, a given employee may be a temporary employee who is a secretary.
- In terms of an E-R diagram, specialization is depicted by a triangle component labeled ISA. The label ISA stands for “is a” and represents, for example, that a customer “is a” person. The ISA relationship may also be referred to as a super class-

subclass relationship. Higher- and lower-level entity sets are depicted as regular entity sets—that is, as rectangles containing the name of the entity set.

2.8.2 Generalization:

- The refinement from an initial entity set into successive levels of entity sub groupings 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 super class and subclass, respectively. The person entity set is the super class of the customer and employee subclasses.
- For all practical purposes, generalization is a simple inversion of specialization. We will apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation will be distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database.

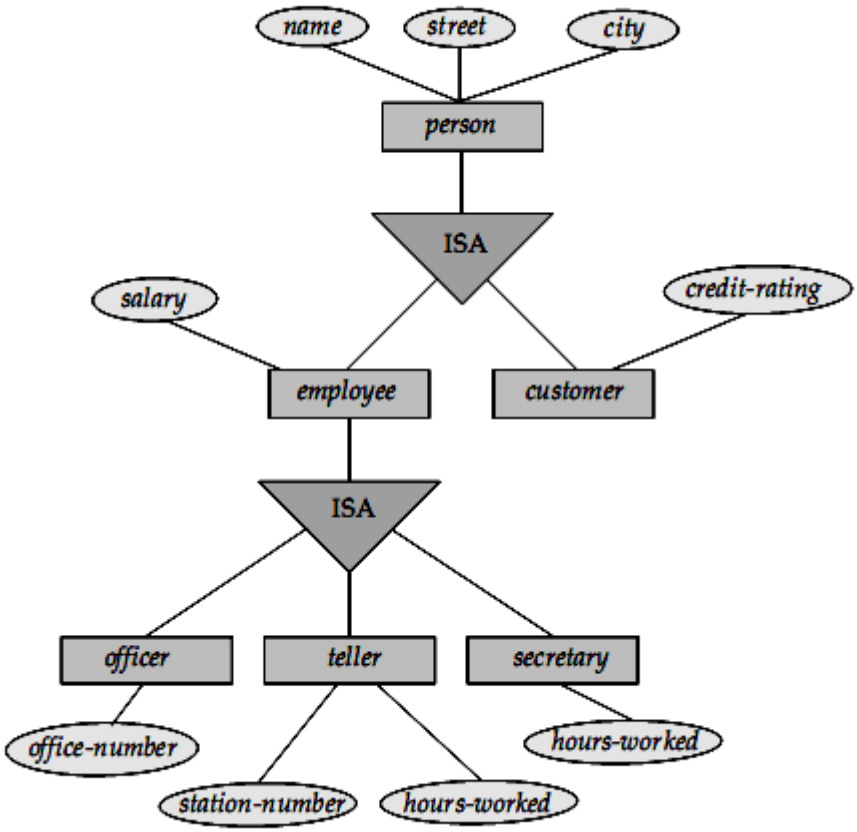


Figure 2.17 Specialization and generalization.

2.8.3 Aggregation:

- One limitation of the E-R model is that it cannot express relationships among relationships.
- To illustrate the need for such a construct, consider the ternary relationship works-on, which we saw earlier, between a employee, branch,and job.
- Now, suppose we want to record managers for tasks performed by an employee at a branch; that is, we want to record managers for (employee, branch, job)combinations. Let us assume that there is an entity set manager.
- One alternative for representing this relationship is to create a quaternary relationship manages between employee, branch, job, and manager. (A quaternary relationship is required—a binary relationship between manager and employee would not permit us to represent which (branch, job) combinations of an employee are managed by which manager.)
- Using the basic E-R modeling constructs, we obtain the E-R diagram as follows:

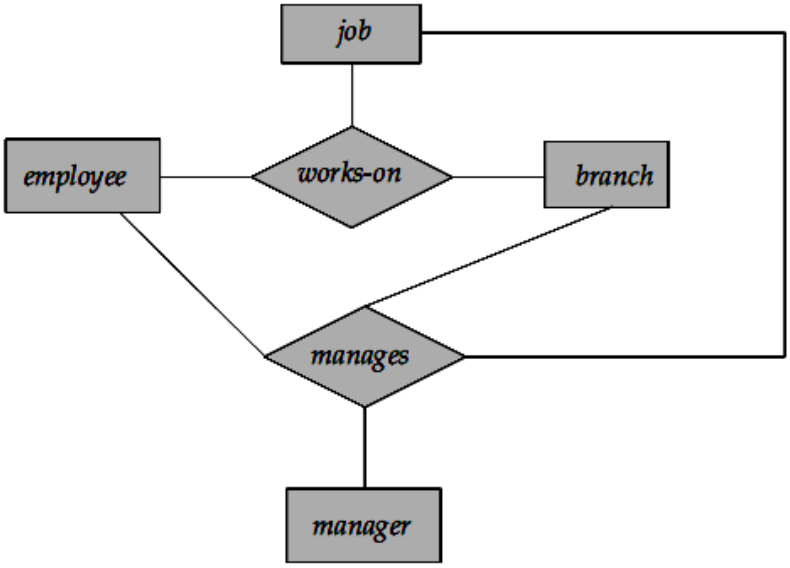


Figure: E-R diagram with redundant relationships.

- It appears that the relationship sets works-on and manages can be combined into one single relationship set.
- Nevertheless, we should not combine them into a single relationship, since some employee, branch, job combinations many not have a manager.
- There is redundant information in the resultant figure, however, since every employee, branch, job combination in manages is also in works-on.
- If the manager were a value rather than an manager entity, we could instead make manager a multi valued attribute of the relationship works-on.
- But doing so makes it more difficult (logically as well as in execution cost) to find, for example, employee-branch-job triples for which a manager is responsible. Since the manager is a manager entity, this alternative is ruled out in any case.
- The best way to model a situation such as the one just described is to use aggregation.
- Aggregation is an abstraction through which relationships are treated as higher-level entities.
- Following figure shows a notation for aggregation commonly used to represent the above situation.

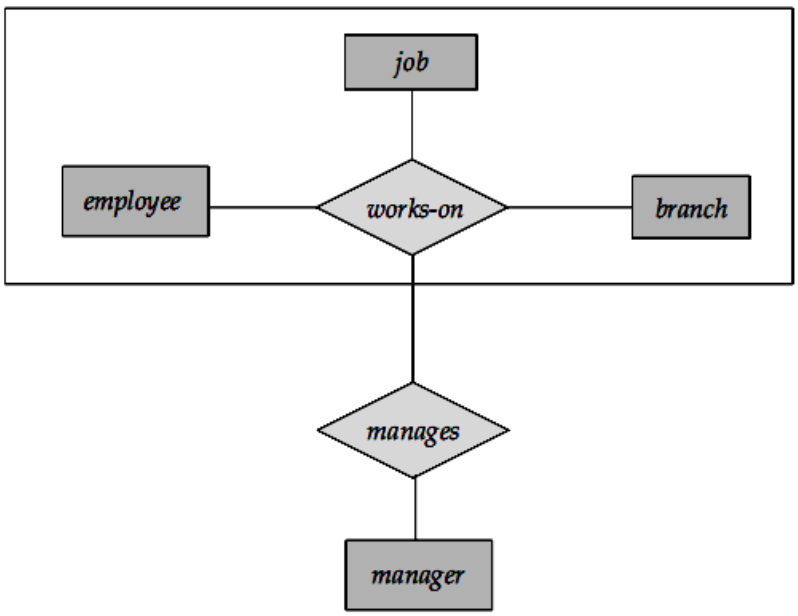


Figure: E-R diagram with aggregation.

2.9 CONCEPTUAL DESIGN WITH E-R MODEL

- 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 multi valued 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 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 the 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.

- 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.

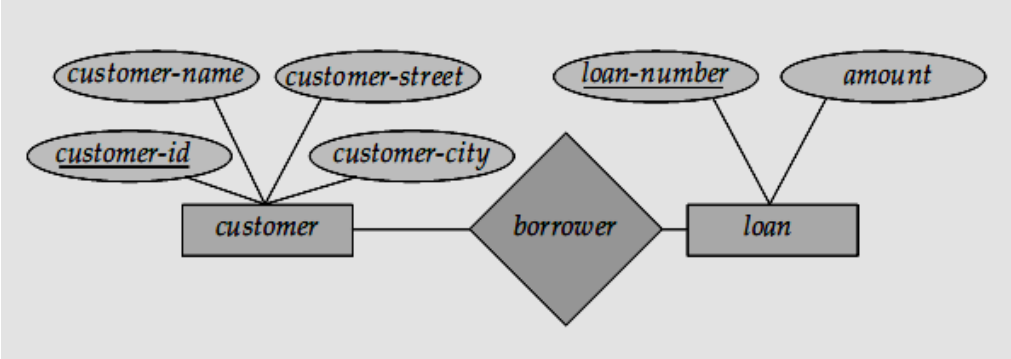


Figure: E-R diagram corresponding to customers and loans.

- If a relationship set has also some attributes associated with it, then we link these attributes to that relationship set. Following figure shows how composite attributes can be represented in the E-R notation.

- Here, a composite attribute name, with component attributes first-name, middle-initial, and last-name replaces the simple attribute customer-name of customer. Also, a composite attribute address, whose component attributes are street, city, state, and zip-code replaces the attributes customer-street and customer-city of customer. The attribute street is itself a composite attribute whose component attributes are street-number, street-name, and apartment number.

- Figure also illustrates a multi valued attribute phone-number, depicted by a double ellipse, and a derived attribute age, depicted by a dashed ellipse.

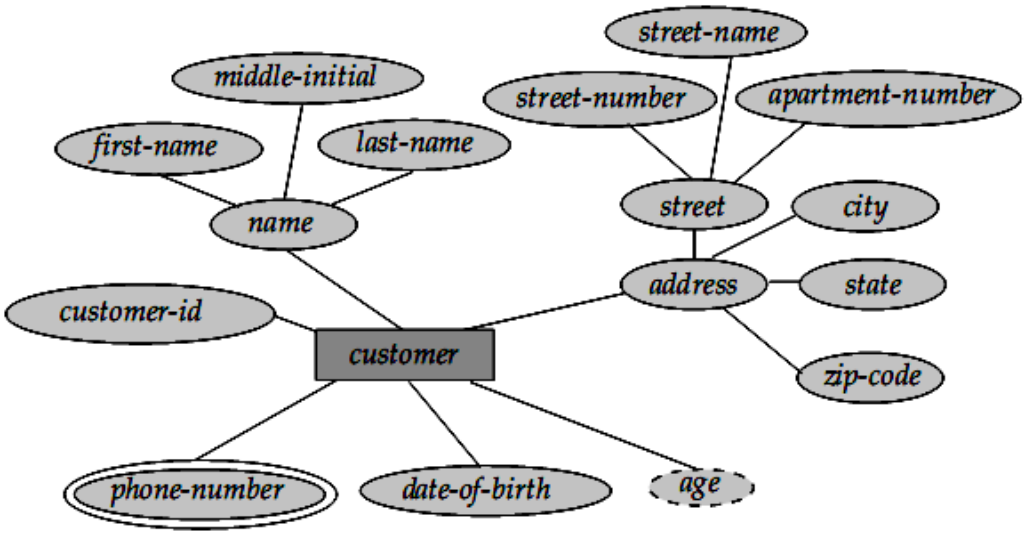
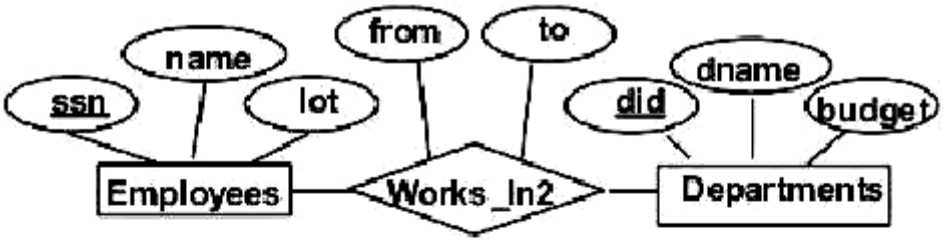
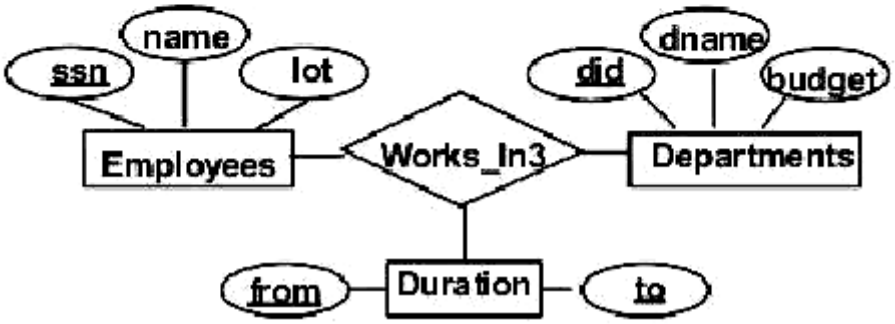


Figure: E-R diagram with composite, multi valued, and derived attributes.

2.10 ENTITY v/s ATTRIBUTE

- Should address be an attribute of Employees or an entity (connected to Employees by a relationship)?
- Depends upon the use we want to make of address information, and the semantics of the data:
    - If we have several addresses per employee, address must be an entity (since attributes cannot be set-valued).
    - If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, address must be modelled as an entity (since attribute values are atomic).
- Works_In2 does not allow an employee to work in a department for two or more periods.
- Similar to the problem of wanting to record several addresses for an employee: we want to record several values of the descriptive attributes for each instance of this relationship.





An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

• We have to record more than one address for an employee.

• We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as "Find all employees with an address in Madison, WI."

For another example of when to model a concept as an entity set rather than an attribute, consider the relationship set shown in following diagram:

Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics, because a relationship is uniquely identified by the participating entities. The problem is that we want to record several values for the descriptive attributes for each instance of the Works-In2

relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes from and to, as shown in following Figure:

2.10 ENTITY v/s RELATIONSHIP

- Suppose that each department manager is given a discretionary budget (dbudget), as shown in following Figure, in which we have also renamed the relationship set to Manages2.

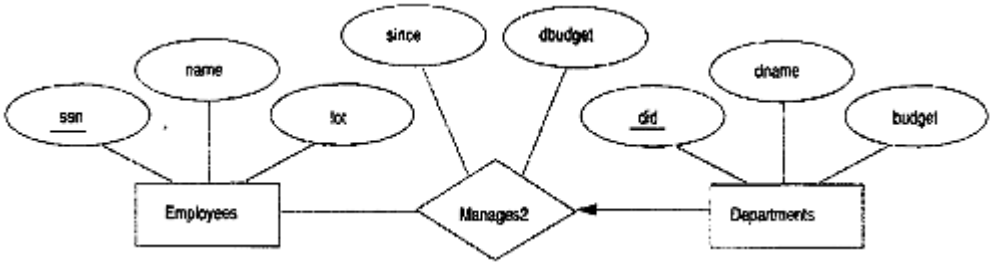


Figure: Entity versus Relationship

- Given a department, we know the manager, as well as the manager's starting date and budge for that department.
- This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.
- But what if the discretionary budget is a sum that covers all departments managed by that employee?
- In this case, each Manages2 relationship that involves a given employee will have the same value in the db1Ldget field, leading to redundant storage of the same information. Another problem with this design is that it is misleading; it suggests that the budget is associated with the relationship, when it is actually associated with the manager.
- We can address these problems by introducing a new entity set called Managers (which can be placed below Employees in an ISA hierarchy, to show that every manager is also an employee).
- The attributes since and dbudget now describe a manager entity, as intended. As a variation, while every manager has a budget, each manager may have a different starting date (as manager) for each department. In this case dbudget is an attribute of Managers, but since is an attribute of the relationship set between managers and departments.
- The imprecise nature of ER modeling can thus make it difficult to recognize underlying entities, and we might associate

attributes with relationships rather than the appropriate entities. In general, such mistakes lead to redundant storage of the same information and can cause many problems.

2.11 BINARY v/s TERNARY RELATIONSHIP

- Consider the ER diagram shown in following Figure. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies. Suppose that we have the following additional requirements:

  ⇨ A policy cannot be owned jointly by two or more employees.
  ⇨ Every policy must be owned by some employee.
  ⇨ Dependents is a weak entity set, and each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity (which, intuitively, covers the given dependent).

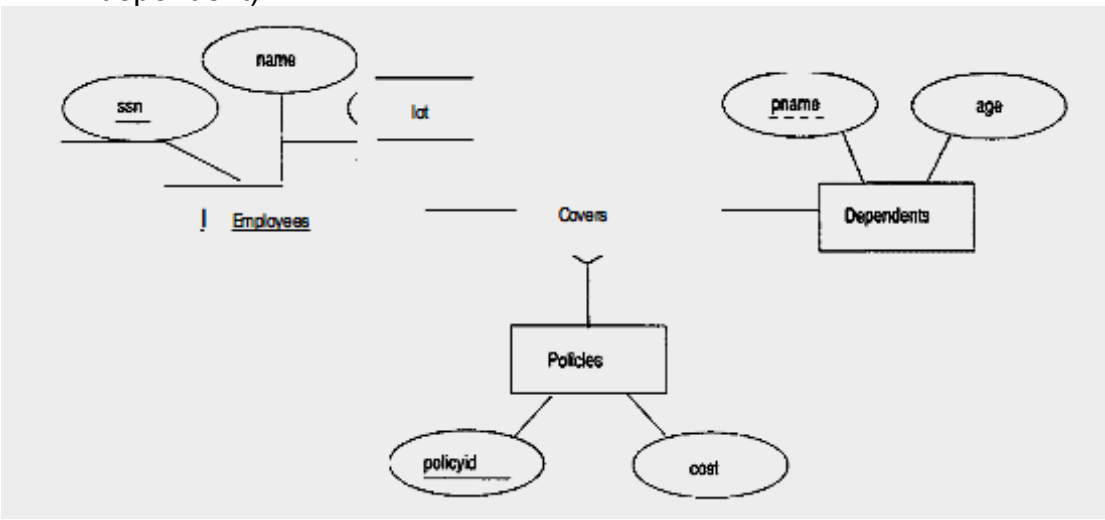


Figure: Policies as an Entity Set

- The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).
  - o Even ignoring the third requirement, the best way to model this situation is to use two binary relationships, as shown in following Figure:

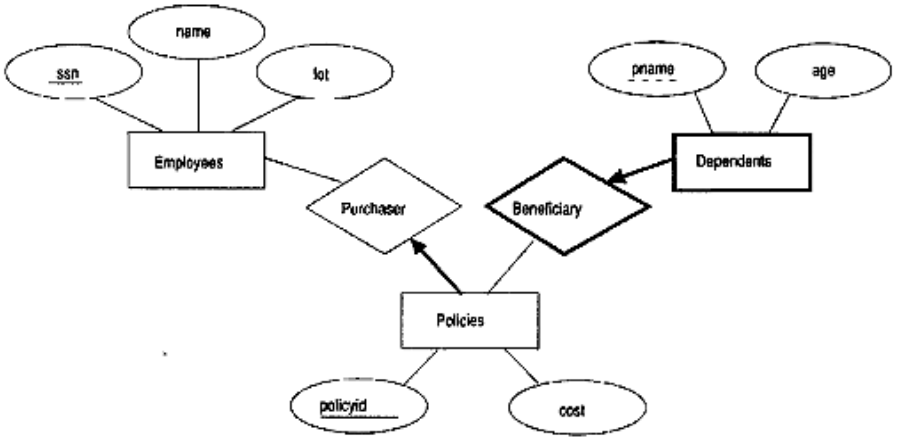


Figure: Policy Revisited

- This example really has two relationships involving Policies, and our attempt to use a single ternary relationship is inappropriate. There are situations, however, where a relationship inherently a.'3sociates more than two entities.

- As a typical example of a ternary relationship, consider entity sets Parts, Sup- pliers, and Departments, and a relationship set Contracts (with descriptive attribute qty) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier 'can supply' certain parts, that a department 'needs' some parts, or that a department 'deals with' a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

• The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S.

• We cannot represent the qty attribute of a contract cleanly.

2.12 AGGREGATE v/s TERNARY RELATIONSHIP

- The choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is

monitored by one or more employees. If we don't need to record the until attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in following Figure.

- Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship Monitors. Thus, the presence of such a constraint serves &s another reason for using aggregation rather than a ternary relationship set.

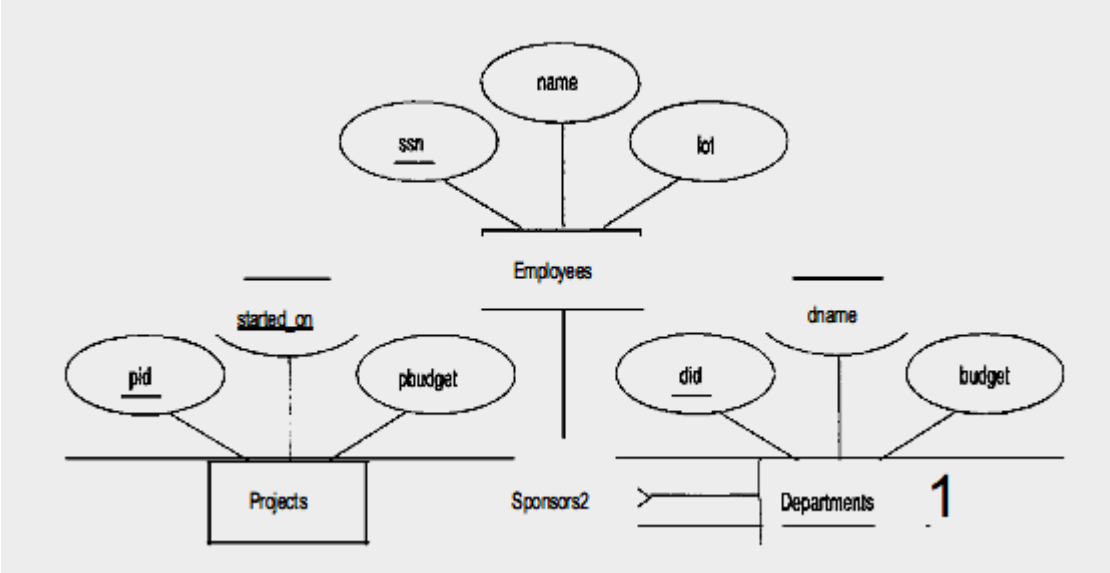


Figure: Using a Ternary Relationship instead of Aggregation

Summary:

- Conceptual design follows requirements analysis,
  - Yields a high-level description of data to be stored
- ER model popular for conceptual design
  - Constructs are expressive, close to the way people think about their applications.
- Basic constructs: entities, relationships, and attributes(of entities and relationships).
- Some additional constructs: weak entities, ISA hierarchies, And aggregation.
- Several kinds of integrity constraints can be expressed in the ER model: *key constraints*, *participation constraints*, and *overlap/covering constraints* for ISA hierarchies.

Some *foreign key constraints* are also implicit in the definition of a relationship set.

- Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model.
- Constraints play an important role in determining the best database design for an enterprise.
- ER design is *subjective*. There are often many ways to model a given scenario! Analyzing alternatives can be tricky, especially for a large enterprise. Common choices include:
    - Entity vs. attribute, entity vs. relationship, binary or n-ary relationship, whether or not to use ISA hierarchies, and whether or not to use aggregation.
- To ensure good database design, resulting relational schema should be analyzed and refined further. FD information and normalization techniques are especially useful.

3

RELATIONAL MODEL

Topics covered

3.1 Introduction to Relational Model

3.2 Creating and modifying Relations using SQL

3.3 Integrity constraints over the Relation

3.4 Logical Database Design: ER to Relational

3.5 Relational Algebra

3.1 INTRODUCTION TO RELATIONAL MODEL:

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent, a "flat" file of records. When a relation is thought of as a table of values, each row in the table represents a collection of related data values. In the relational model, each row in the table represents a fact that typically corresponds to a real world entity or relationship. The table name and column names are used to help in interpreting the meaning of the values in each row. In the formal relational model terminology, a row is called a tuple, a column header is called an attribute, and the table is called a relation. The data type describing the types of values that can appear in each column is called a domain. We now define these terms—domain, tuple, attribute, and relation—more precisely.

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figure: The account relation.

3.2 CREATING AND MODIFYING RELATIONS USING SQL

3.2.1 Creating Relations: (CREATE TABLE STATEMENT)

The CREATE TABLE statement, defines a new table(Relation) in the database and prepares it to accept data. The various clauses of the statement specify the elements of the table definition.

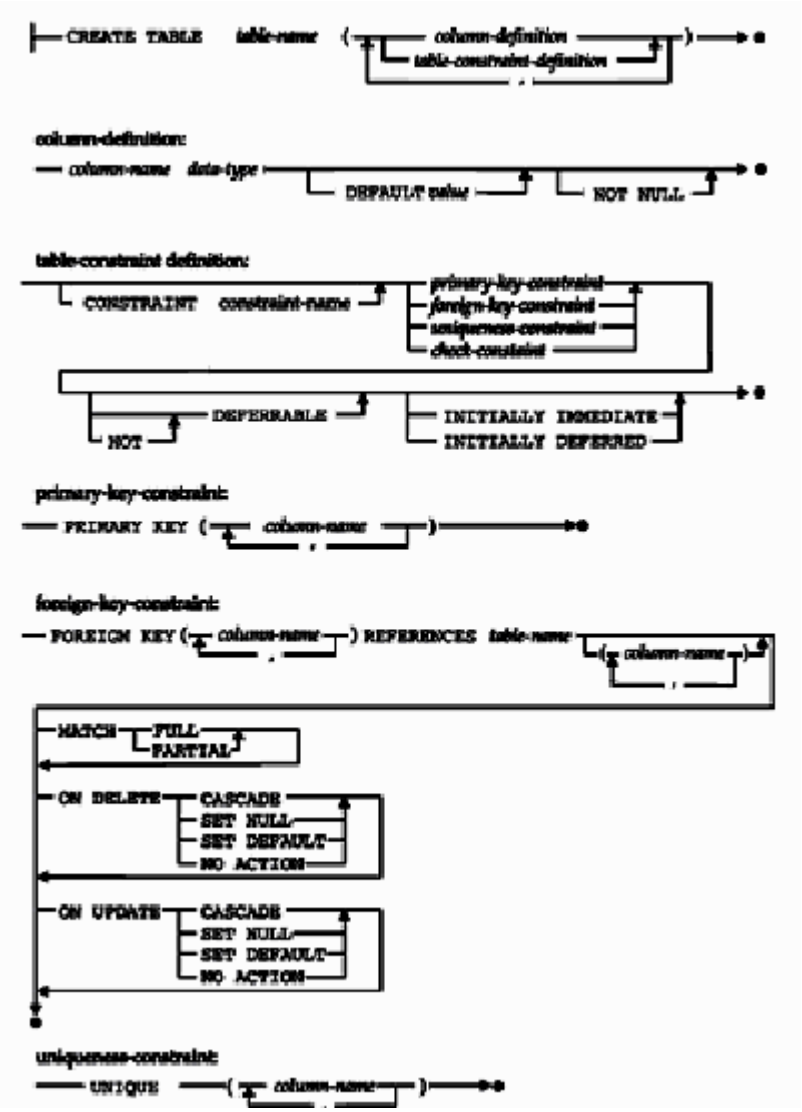


Figure: Basic CREATE TABLE syntax diagram

SQL CREATE TABLE statement defines a new table to store the products data:

```
CREATE TABLE PRODUCTS
    (MFR_ID CHAR(3),
  PRODUCT_ID CHAR(5),
DESCRIPTION VARCHAR(20),
     PRICE MONEY,
QTY_ON_HAND INTEGER)

Table created
```

Although more cryptic than the previous SQL statements, the CREATE TABLE statement is still fairly straightforward. It assigns the name PRODUCTS to the new table and specifies the name and type of data stored in each of its five columns.

Once the table has been created, you can fill it with data.

3.2.1 Modifying Relations: (ALTER TABLE STATEMENT)

After a table has been in use for some time, users often discover that they want to store additional information about the entities represented in the table.

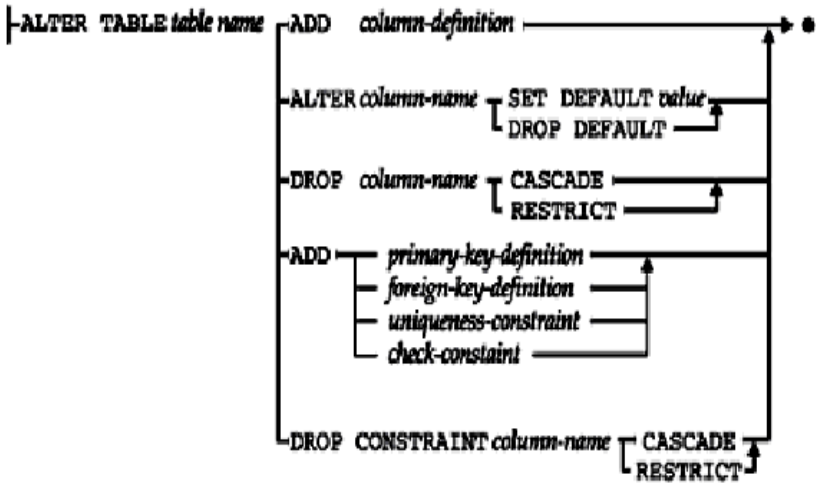


Figure : ALTER TABLE statement syntax diagram

The ALTER TABLE statement can:

- Add a column definition to a table
- Drop a column from a table
- Change the default value for a column
- Add or drop a primary key for a table
- Add or drop a new foreign key for a table
- Add or drop a uniqueness constraint for a table
- Add or drop a check constraint for a table.

For example: Add a minimum inventory level column to the PRODUCTS table.

```
ALTER TABLE PRODUCTS
  ADD MIN_QTY INTEGER NOT NULL WITH DEFAULT 0
```

In the first example, the new columns will have NULL values for existing customers. In the second example, the MIN_QTY column will have the value zero (0) for existing products, which is appropriate.

3.3 INTEGRITY CONSTRAINTS OVER THE RELATION:

To preserve the consistency and correctness of its stored data, a relational DBMS typically imposes one or more data integrity constraints. These constraints restrict the data values that can be inserted into the database or created by a database update. Several different types of data integrity constraints are commonly found in relational databases, including:

• **Required data**: Some columns in a database must contain a valid data value in every row; they are not allowed to contain missing or NULL values. In the sample database, every order must have an associated customer who placed the order. The DBMS can be asked to prevent NULL values in this column.

• **Validity checking**: Every column in a database has a domain, a set of data values that are legal for that column. The DBMS can be asked to prevent other data values in these columns.

• **Entity integrity**: The primary key of a table must contain a unique value in each row, which is different from the values in all other rows. Duplicate values are illegal, because they wouldn't allow the database to distinguish one entity from another. The DBMS can be asked to enforce this unique values constraint.

• **Referential integrity**: A foreign key in a relational database links each row in the child table containing the foreign key to the row of the parent table containing the matching primary key value. The DBMS can be asked to enforce this foreign key/primary key constraint.

• **Other data relationships**: The real-world situation modeled by a database will often have additional constraints that govern the legal data values that may appear in the database.

The DBMS can be asked to check modifications to the tables to make sure that their values are constrained in this way.

• **Business rules:** Updates to a database may be constrained by business rules governing the real-world transactions that are represented by the updates.

• **Consistency**: Many real-world transactions cause multiple updates to a database. The DBMS can be asked to enforce this type of consistency rule or to support applications that implement such rules.

3.4 LOGICAL DATABASE DESIGN: ER TO RELATIONAL

The ER model is convenient for representing an initial, high-level database design. Given an ER diagram describing a databa'3e, a standard approach is taken to generating a relational database schema that closely approximates the ER design. (The translation is approximate to the extent that we cannot capture all the constraints implicit in the ER design using SQL, unless we use certain SQL constraints that are costly to check.) We now describe how to translate an ER diagram into a collection of tables with associated constraints, that is, a relational database schema.

3.4.1 Entity Sets to Tables

An entity set is mapped to a relation in a straightforward way: Each attribute of the entity set becomes an attribute of the table. Note that we know both the domain of each attribute and the (primary) key of an entity set. Consider the Employees entity set with attributes ssn, name, and lot shown in following Figure.

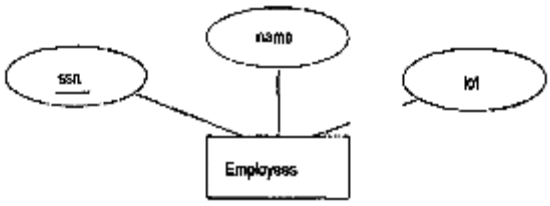


Figure: The Employees Entity Set

A possible instance of the Employees entity set, containing three Employees entities, is shown in following Figure in a tabular format.

ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

Figure: An Instance of the Employees Entity Set

3.5 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. We will define these operations in terms of the fundamental operations.

3.5.1 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 operations.

3.5.1.1 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

σbranch-name ="Perryridge" (loan)

We can find all tuples in which the amount lent is more than $1200 by writing σamount>1200 (loan)

In general, we allow comparisons using =, □

=, <, ≤, >, ≥ in the selection predicate.

Furthermore, we can combine several predicates into a larger predicate by using the connectives and (∧), or (∨), and not (¬). Thus, to find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write:

σbranch-name ="Perryridge"∧ amount>1200 (loan)

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

Figure: Result of σbranch-name ="Perryridge" (loan).

The selection predicate may include comparisons between two attributes. To illustrate, consider the relation loan-officer that consists of three attributes: customer-name, banker-name, and loan-number, which specifies that a particular banker is the loan officer for a loan that belongs to some customer. To find all customers who have the same name as their loan officer, we can write σcustomer-name =banker-name (loan-officer).

Relational algebra, an offshoot of first-order logic (and of algebra of sets), deals with a set of finitary relations which is closed under certain operators. These operators operate on one or more relations to yield a relation.

As in any algebra, some operators are primitive and the others, being definable in terms of the primitive ones, are derived. It is useful if the choice of primitive operators parallels the usual choice of primitive logical operators. Although it is well known that the usual choice in logic of AND, OR and NOT is somewhat arbitrary, Codd made a similar arbitrary choice for his algebra.

The six primitive operators of Codd's algebra are the *selection*, the *projection*, the *Cartesian product* (also called the *cross product* or *cross join*), the *set union*, the *set difference*, and the *rename*. (Actually, Codd omitted the rename, but the compelling case for its inclusion was shown by the inventors of ISBL.) These six operators are fundamental in the sense that none of them can be omitted without losing expressive power. Many other operators have been defined in terms of these six. Among the most important are set intersection, division, and the natural join. In fact ISBL made a compelling case for replacing the Cartesian product with the natural join, of which the Cartesian product is a degenerate case.

Altogether, the operators of relational algebra have identical expressive power to that of domain relational calculus or tuple relational calculus. However, for the reasons given in the Introduction above, relational algebra has strictly less expressive power than that of first-order predicate calculus without function symbols. Relational algebra actually corresponds to a subset of first-order logic that is Horn clauses *without* recursion and negation.

Set operators

Although three of the six basic operators are taken from set theory, there are additional constraints that are present in their relational algebra counterparts: For set union and set difference, the two relations involved must be *union-compatible*—that is, the two relations must have the same set of attributes. As set intersection can be defined in terms of set difference, the two relations involved in set intersection must also be union-compatible.

The Cartesian product is defined differently from the one defined in set theory in the sense that tuples are considered to be 'shallow' for the purposes of the operation. That is, unlike in set theory, where the Cartesian product of a *n*-tuple by an *m*-tuple is a set of 2-tuples, the Cartesian product in relational algebra has the 2-tuple "flattened" into an *n*+*m*-tuple. More formally, $R \times S$ is defined as follows:

$$R \times S = \{r \cup s \mid r \in R, s \in S\}$$

In addition, for the Cartesian product to be defined, the two relations involved must have disjoint headers — that is, they must not have a common attribute name.

Projection (π)

A **projection** is a unary operation written as $\pi_{a_1,\ldots,a_n}(R)$ where $a_1,\ldots,a_n$ is a set of attribute names. The result of such projection is defined as the set that is obtained when all tuples in *R* are restricted to the set $\{a_1,\ldots,a_n\}$.

Selection (σ)

A **generalized selection** is a unary operation written as $\sigma_{\varphi}(R)$ where φ is a propositional formula that consists of atoms as allowed in the normal selection and the logical operators $\wedge$ (and), $\vee$ (or) and $\neg$ (negation). This selection selects all those tuples in *R* for which φ holds.

Rename (ρ)

A **rename** is a unary operation written as $\rho_{a\,/\,b}(R)$ where the result is identical to *R* except that the *b* field in all tuples is renamed to an *a* field. This is simply used to rename the attribute of a relation or the relation itself.

4

SQL

Topics covered

4.1 Data Definition Commands
4.2 Constraints
4.3 View
4.4 Data Manipulation Commands
4.5 Queries
4.6 Aggregate Queries
4.7 NULL values
4.8 Outer Joins
4.9 Nested Queries- Correlated Queries
4.10 Embedded SQL
4.11 Dynamic SQL
4.12 TRIGGERS

4.1 DATA DEFINITION COMMANDS:

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

4.1.1 Create Table Statement

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.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain consistency constraints. For example, suppose the balance on an account should not fall below $100. The DDL provides facilities to specify such constraints. The database systems check these constraints every time the database is updated.

4.1.2 DROP table statement:

Over time the structure of a database grows and changes. New tables are created to represent new entities, and some old tables are no longer needed. You can remove an unneeded table from the database with the DROP TABLE statement

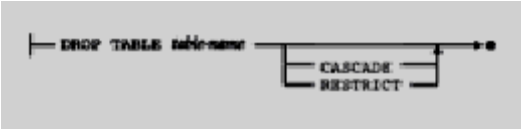


Figure 13-3: DROP TABLE statement syntax diagram

The table name in the statement identifies the table to be dropped. Normally you will be dropping one of your own tables and will use an unqualified table name. With proper permission, you can also drop a table owned by another user by specifying a qualified table name.

For example:
DROP TABLE CUSTOMER

4.1.3 ALTER Table statement

Refer to the section: **3.2.1 Modifying Relations: (ALTER TABLE STATEMENT)**

4.2 CONSTRAINTS

A SQL2 check constraint is a search condition, like the search condition in a WHERE clause, that produces a true/false value. When a check constraint is specified for a column, the DBMS automatically checks the value of that column each time a new row is inserted or a row is updated to insure that the search condition is true. If not, the INSERT or UPDATE statement fails. A column check constraint is specified as part of the column definition within the CREATE TABLE statement.

Consider this excerpt from a CREATE TABLE statement, includes three check constraints:

```
CREATE TABLE SALESREPS
(EMPL_NUM INTEGER NOT NULL
        CHECK (EMPL_NUM BETWEEN 101 AND 199),
    AGE INTEGER
        CHECK (AGE >= 21),
  QUOTA MONEY
        CHECK (MONEY >= 0.0)     )
```

The first constraint (on the EMPL_NUM column) requires that valid employee numbers be three-digit numbers between 101 and 199. The second constraint (on the AGE column) similarly prevents hiring of minors. The third constraint (on the QUOTA column) prevents a salesperson from having a quota target less than $0.00.

All three of these column check constraints are very simple examples of the capability specified by the SQL2 standard. In general, the parentheses following the keyword CHECK can contain any valid search condition that makes sense in the context of a column definition. With this flexibility, a check constraint can compare values from two different columns of the table, or even compare a proposed data value against other values from the database.

4.3 VIEW

A view is a "virtual table" in the database whose contents are defined by a query.

The tables of a database define the structure and organization of its data. However, SQL also lets you look at the stored data in other ways by defining alternative views of the data. A view is a SQL query that is permanently stored in the database and assigned a name. The results of the stored query are "visible" through the view, and SQL lets you access these query results as if they were, in fact, a "real" table in the database.

Views are an important part of SQL, for several reasons:

- Views let you tailor the appearance of a database so that different users see it from different perspectives.
- Views let you restrict access to data, allowing different users to see only certain rows or certain columns of a table.
- Views simplify database access by presenting the structure of the stored data in the way that is most natural for each user.

4.3.1 Advantages of VIEW

Views provide a variety of benefits and can be useful in many different types of databases. In a personal computer database, views are usually a convenience, defined to simplify database requests. In a production database installation, views play a central role in defining the structure of the database for its users and enforcing its security. Views provide these major benefits:

• **Security:** Each user can be given permission to access the database only through a small set of views that contain the specific data the user is authorized to see, thus restricting the user's access to stored data.

• **Query simplicity:** A view can draw data from several different tables and present it as a single table, turning multi-table queries into single-table queries against the view.

• **Structural simplicity:** Views can give a user a "personalized" view of the database structure, presenting the database as a set of virtual tables that make sense for that user.

• **Insulation from change:** A view can present a consistent, unchanged image of the structure of the database, even if the underlying source tables are split, restructured, or renamed.

• **Data integrity:** If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

4.3.2 Disadvantages of VIEW

While views provide substantial advantages, there are also two major disadvantages to using a view instead of a real table:

• **Performance:** Views create the appearance of a table, but the DBMS must still translate queries against the view into queries against the underlying source tables. If the view is defined by a complex, multi-table query, then even a simple query against the view becomes a complicated join, and it may take a long time to complete.

• **Update restrictions:** When a user tries to update rows of a view, the DBMS must translate the request into an update on rows of the underlying source tables. This is possible for simple views, but more complex views cannot be updated; they are "read-only."

These disadvantages mean that you cannot indiscriminately define views and use them instead of the source tables. Instead, you must in each case consider the advantages provided by using a view and weigh them against the disadvantages.

4.3.3 Creating a VIEW

The CREATE VIEW statement is used to create a view. The statement assigns a name to the view and specifies the query that defines the view. To create the view successfully, you must have permission to access all of the tables referenced in the query.

The CREATE VIEW statement can optionally assign a name to each column in the newly created view. If a list of column names is specified, it must have the same number of items as the number of columns produced by the query. Note that only the column names are specified; the data type, length, and other characteristics of each column are derived from the definition of the columns in the source tables. If the list of column names is omitted from the CREATE VIEW statement, each column in the view takes the name of the corresponding column in the query. The list of column names must be specified if the query includes calculated columns or if it produces two columns with identical names.

For example:
Define a view containing only Eastern region offices.

```
CREATE VIEW EASTOFFICES AS
    SELECT *
      FROM OFFICES
     WHERE REGION = 'Eastern'
```

4.4 DATA MANIPULATION COMMANDS

DML Commands are used for manipulating data in database.

4.4.1 Insert Statement

The INSERT statement, adds a new row to a table. The INTO clause specifies the table that receives the new row (the target table), and the VALUES clause specifies the data values that the new row will contain. The column list indicates which data value goes into which column of the new row.

For example:

```
INSERT INTO SALESREPS(NAME, AGE, EMPL_NUM, SALES,
TITLE,
      HIRE_DATE, REP_OFFICE)
     VALUES ('Henry Jacobsen', 36, 111, 0.00, 'Sales Mgr',
                  '25-JUL-90', 13)
  1 row inserted.
```

The INSERT statement builds a single row of data that matches the column structure of the table, fills it with the data from the VALUES clause, and then adds the new row to the table. The rows of a table are unordered, so there is no notion of inserting the

row "at the top" or "at the bottom" or "between two rows" of the table. After the INSERT statement, the new row is simply a part of the table. A subsequent query against the SALESREPS table will include the new row, but it may appear anywhere among the rows of query results.

4.4.2 Delete Statement

The DELETE statement removes selected rows of data from a single table. The FROM clause specifies the target table containing the rows. The WHERE clause specifies which rows of the table are to be deleted.
For example:
Remove Henry Jacobsen from the database.

```
DELETE FROM SALESREPS
WHERE NAME = 'Henry Jacobsen'

1 row deleted.
```

The WHERE clause in this example identifies a single row of the SALESREPS table, which SQL removes from the table.

We can delete all the rows from a table.

For example:

```
DELETE FROM ORDERS

30 rows deleted.
```

4.4.3 Update Statement

The UPDATE statement modifies the values of one or more columns in selected rows of a single table. The target table to be updated is named in the statement, and you must have the required permission to update the table as well as each of the individual columns that will be modified. The WHERE clause selects the rows of the table to be modified. The SET clause specifies which columns are to be updated and calculates the new values for them.
For example:

Here is a simple UPDATE statement that changes the credit limit and salesperson for a customer:

Raise the credit limit for Acme Manufacturing to $60,000 and reassign them to Mary Jones (employee number 109).

```
UPDATE CUSTOMERS
SET CREDIT_LIMIT = 60000.00, CUST_REP = 109
WHERE COMPANY = 'Acme Mfg.'

1 row updated.
```

In this example, the WHERE clause identifies a single row of the CUSTOMERS table, and the SET clause assigns new values to two of the columns in that row.

4.5 QUERIES

Select-From-Where Statements

The SELECT statement retrieves data from a database and returns it to you in the form of query results.

The SELECT clause lists the data items to be retrieved by the SELECT statement. The items may be columns from the database, or columns to be calculated by SQL as it performs the query.

• The FROM clause lists the tables that contain the data to be retrieved by the query.

• The WHERE clause tells SQL to include only certain rows of data in the query results. A search condition is used to specify the desired rows.

For Example:

```
SELECT NAME, HIRE_DATE
   FROM SALESREPS
  WHERE SALES > 500000.00
```

4.6 AGGREGATE QUERIES

Aggregate functions are functions that take a collection (a set or multi set) 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

⇨ 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’
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average balance at the Perryridge branch.

⇨ Consider the query “Find the **minimum** salary offered to a employee.”We write this query as follows:

```
select min (salary)
From employee
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the minimum salary offered to an employee.

⇨ Consider the query “Find the **maximum** salary offered to a employee.”We write this query as follows:

```
select max (salary)
From employee
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the maximum salary offered to an employee

⇨ To find the number of tuples in the customer relation, we write

```
select count (*)
from customer.
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the total number of customers present in the customer table.

⇨ To find the total salary issued to the employees we write the query:

```
select sum (salary)
from employee
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the addition of the salaries offered to all the employees.

4.7 NULL VALUES:

Because a database is usually a model of a real-world situation, certain pieces of data are inevitably missing, unknown, or don't apply. In the sample database, for example, the QUOTA column in the SALESREPS table contains the sales goal for each salesperson. However, the newest salesperson has not yet been assigned a quota; this data is missing for that row of the table. You might be tempted to put a zero in the column for this salesperson, but that would not be an accurate reflection of the situation. The salesperson does not have a zero quota; the quota is just "not yet known."

Similarly, the MANAGER column in the SALESREPS table contains the employee number of each salesperson's manager. But Sam Clark, the Vice President of Sales, has no manager in the sales organization. This column does not apply to Sam. Again, you might think about entering a zero, or a 9999 in the column, but neither of these values would really be the employee number of Sam's boss. No data value is applicable to this row.

SQL supports missing, unknown, or inapplicable data explicitly, through the concept of a null value. A null value is an indicator that tells SQL (and the user) that the data is missing or not applicable. As a convenience, a missing piece of data is often said to have the value NULL. But the NULL value is not a real data value like 0, 473.83, or "Sam Clark." Instead, it's a signal, or a reminder, that the data value is missing or unknown.

In many situations NULL values require special handling by the DBMS. For example, if the user requests the sum of the QUOTA column, how should the DBMS handle the missing data when computing the sum? The answer is given by a set of special rules that govern NULL value handling in various SQL statements and clauses. Because of these rules, some leading database authorities feel strongly that NULL values should not be used.

4.8 OUTER JOINS

The process of forming pairs of rows by matching the contents of related columns is called joining the tables. The resulting table (containing data from both of the original tables) is called a join between the two tables.

The SQL join operation combines information from two tables by forming pairs of related rows from the two tables. The row pairs that make up the joined table are those where the matching columns in each of the two tables have the same value. If one of the rows of a table is unmatched in this process, the join can produce unexpected results, as illustrated by these queries:

List the salespeople and the offices where they work.

```
SELECT NAME, REP_OFFICE
  FROM SALESREPS

NAME          REP_OFFICE
--------------    ----------
Bill Adams          13
Mary Jones           11
Sue Smith           21
Sam Clark           11
Bob Smith           12
```

```
Dan Roberts          12
Tom Snyder          NULL
Larry Fitch         21
Paul Cruz           12
Nancy Angelli       22
```

List the salespeople and the cities where they work.

```
SELECT NAME, CITY
  FROM SALESREPS, OFFICES
WHERE REP_OFFICE = OFFICE

NAME          CITY
------------  --------
Mary Jones    New York
Sam Clark     New York
Bob Smith     Chicago
Paul Cruz     Chicago
Dan Roberts   Chicago
Bill Adams    Atlanta
Sue Smith     Los Angeles
Larry Fitch   Los Angeles
Nancy Angelli Denver
```

The outer join query that will combine the results of above queries and join the 2 tables is as follows:

List the salespeople and the cities where they work.

```
SELECT NAME, CITY
  FROM SALESREPS, OFFICES
WHERE REP_OFFICE *= OFFICE

NAME          CITY
------------      --------
Tom Snyder    NULL
Mary Jones    New York
Sam Clark     New York
Bob Smith     Chicago
Paul Cruz     Chicago
Dan Roberts   Chicago
Bill Adams    Atlanta
Sue Smith     Los Angeles
Larry Fitch   Los Angeles
Nancy Angelli Denver
```

4.8.1 Left and Right outer join

Technically, the outer join produced by the previous query is called the full outer join of the two tables. Both tables are treated symmetrically in the full outer join. Two other well-defined outer joins do not treat the two tables symmetrically.

The left outer join between two tables is produced by following Step 1 and Step 2 in the previous numbered list but omitting Step 3. The left outer join thus includes NULL-extended copies of the unmatched rows from the first (left) table but does not include any unmatched rows from the second (right) table. Here is a left outer join between the GIRLS and BOYS tables: List girls and boys in the same city and any unmatched girls.

```
SELECT *
  FROM GIRLS, BOYS
WHERE GIRLS.CITY *= BOYS.CITY

GIRLS.NAME  GIRLS.CITY  BOYS.NAME  BOYS.CITY
----------  ----------  ---------  ---------
Mary        Boston      John       Boston
Mary        Boston      Henry      Boston
Susan       Chicago     Sam        Chicago
Betty       Chicago     Sam        Chicago
Anne        Denver      NULL       NULL
Nancy       NULL        NULL       NULL
```

The query produces six rows of query results, showing the matched girl/boy pairs and the unmatched girls. The unmatched boys are missing from the results.

Similarly, the right outer join between two tables is produced by following Step 1 and Step 3 in the previous numbered list but omitting Step 2. The right outer join thus includes NULL-extended copies of the unmatched rows from the second (right) table but does not include the unmatched rows of the first (left) table. Here is a right outer join between the GIRLS and BOYS tables:

List girls and boys in the same city and any unmatched boys.

```
SELECT *
  FROM GIRLS, BOYS
WHERE GIRLS.CITY =* BOYS.CITY

GIRLS.NAME  GIRLS.CITY  BOYS.NAME  BOYS.CITY
----------  ----------  ---------  ---------
Mary        Boston      John       Boston
Mary        Boston      Henry      Boston
Susan       Chicago     Sam        Chicago
```

```
Betty      Chicago    Sam       Chicago
NULL       NULL       James     Dallas
NULL       NULL       George    NULL
```

This query also produces six rows of query results, showing the matched girl/boy pairs and the unmatched boys. This time the unmatched girls are missing from the results.

As noted before, the left and right outer joins do not treat the two joined tables symmetrically. It is often useful to think about one of the tables being the "major" table (the one whose rows are all represented in the query results) and the other table being the "minor" table (the one whose columns contain NULL values in the joined query results). In a left outer join, the left (first-mentioned) table is the major table, and the right (later-named) table is the minor table. The roles are reversed in a right outer join (right table is major, left table is minor).

In practice, the left and right outer joins are more useful than the full outer join, especially when joining data from two tables using a parent/child (primary key/foreign key) relationship. To illustrate, consider once again the sample database. We have already seen one example involving the SALESREPS and OFFICES table. The REP_OFFICE column in the SALESREPS table is a foreign key to the OFFICES table; it tells the office where each salesperson works, and it is allowed to have a NULL value for a new salesperson who has not yet been assigned to an office. Tom Snyder is such a salesperson in the sample database. Any join that exercises this SALESREPS-to-OFFICES relationship and expects to include data for Tom Snyder must be an outer join, with the SALESREPS table as the major table. Here is the example used earlier:

List the salespeople and the cities where they work.

```
SELECT NAME, CITY
  FROM SALESREPS, OFFICES
WHERE REP_OFFICE *= OFFICE

NAME          CITY
------------  --------
Tom Snyder     NULL
Mary Jones     New York
Sam Clark      New York
Bob Smith      Chicago
Paul Cruz      Chicago
Dan Roberts    Chicago
Bill Adams     Atlanta
```

```
Sue Smith      Los Angeles
Larry Fitch    Los Angeles
Nancy Angelli  Denver
```

Note in this case (a left outer join), the "child" table (SALESREPS, the table with the foreign key) is the major table in the outer join, and the "parent" table (OFFICES) is the minor table. The objective is to retain rows containing NULL foreign key values (like Tom Snyder's) from the child table in the query results, so the child table becomes the major table in the outer join. It doesn't matter whether the query is actually expressed as a left outer join (as it was previously) or as a right outer join like this:

List the salespeople and the cities where they work.

```
SELECT NAME, CITY
  FROM SALESREPS, OFFICES
WHERE OFFICE =* REP_OFFICE
 NAME          CITY
-------------  ---------
Tom Snyder     NULL
Mary Jones     New York
Sam Clark      New York
Bob Smith      Chicago
Paul Cruz      Chicago
Dan Roberts    Chicago
Bill Adams     Atlanta
Sue Smith      Los Angeles
Larry Fitch    Los Angeles
Nancy Angelli  Denver
```

What matters is that the child table is the major table in the outer join.

There are also useful joined queries where the parent is the major table and the child table is the minor table. For example, suppose the company in the sample database opens a new sales office in Dallas, but initially the office has no salespeople assigned to it. If you want to generate a report listing all of the offices and the names of the salespeople who work there, you might want to include a row representing the Dallas office. Here is the outer join query that produces those results:

List the offices and the salespeople who work in each one.

```
SELECT CITY, NAME
  FROM OFFICES, SALESREPS
WHERE OFFICE *= REP_OFFICE
```

CITY	NAME
New York	Mary Jones
New York	Sam Clark
Chicago	Bob Smith
Chicago	Paul Cruz
Chicago	Dan Roberts
Atlanta	Bill Adams
Los Angeles	Sue Smith
Los Angeles	Larry Fitch
Denver	Nancy Angelli
Dallas	NULL

In this case, the parent table (OFFICES) is the major table in the outer join, and the child table (SALESREPS) is the minor table. The objective is to insure that all rows from the OFFICES table are represented in the query results, so it plays the role of major table. The roles of the two tables are precisely reversed from the previous example. Of course, the row for Tom Snyder, which was included in the query results for the earlier example (when SALESREPS was the major table), is missing from this set of query results because SALESREPS is now the minor table.

4.9 NESTED QUERIES- CORRELATED QUERIES

4.9.1 Nested Queries (Sub Queries)

A subquery or a nested query is a query-within-a-query. The results of the subquery are used by the DBMS to determine the results of the higher-level query that contains the subquery. In the simplest forms of a subquery, the subquery appears within the WHERE or HAVING clause of another SQL statement. Sub queries provide an efficient, natural way to handle query requests that are themselves expressed in terms of the results of other queries. Here is an example of such a request:

List the offices where the sales target for the office exceeds the sum of the individual salespeople's quotas.

The request asks for a list of offices from the OFFICES table, where the value of the TARGET column meets some condition. It seems reasonable that the SELECT statement that expresses the query should look something like this:

```
SELECT CITY
 FROM OFFICES
WHERE TARGET >???
```

The value "???" needs to be filled in and should be equal to "the sum of the quotas of the salespeople assigned to the office in question." How can you specify that value in the query? Thesum of the quotas for a specific office (say, office number 21) can be obtained with this query:

```
  SELECT SUM(QUOTA)
  FROM SALESREPS
WHERE REP_OFFICE = 21
```

But how can you put the results of this query into the earlier query in place of the question marks? It would seem reasonable to start with the first query and replace the "???" with the second query, as follows:

```
  SELECT CITY
  FROM OFFICES
WHERE TARGET > (SELECT SUM (QUOTA)
                FROM SALESREPS
               WHERE REP_OFFICE = OFFICE)
```

⇨ A few differences between a nested query or subquery and an actual SELECT statement:

- In the most common uses, a nested query or subquery must produce a single column of data as its query results. This means that a subquery almost always has a single select item in its SELECT clause.
- The ORDER BY clause cannot be specified in a nested query or subquery. The subquery results are used internally by the main query and are never visible to the user, so it makes little sense to sort them anyway.
- Column names appearing in a subquery may refer to columns of tables in the main query.
- In most implementations, a subquery cannot be the UNION of several different SELECT statements; only a single SELECT is allowed. (The SQL2 standard allows much more powerful query expressions and relaxes this restriction

4.9.2 Correlated Queries:

In concept, SQL performs a subquery over and over again—once for each row of the main query. For many sub queries, however, the subquery produces the same results for every row or row group. Here is an example:

List the sales offices whose sales are below the average target.

```
 SELECT CITY
 FROM OFFICES
WHERE SALES < (SELECT AVG(TARGET)
               FROM OFFICES)
 CITY
-------
Denver
Atlanta
```

In this query, it would be silly to perform the subquery five times (once for each office). The average target doesn't change with each office; it's completely independent of the office currently being tested. As a result, SQL can handle the query by first performing the subquery, yielding the average target ($550,000), and then converting the main query into:

```
  SELECT CITY
    FROM OFFICES
  WHERE SALES < 550000.00
```

Commercial SQL implementations automatically detect this situation and use this shortcut whenever possible to reduce the amount of processing required by a subquery. However, the shortcut cannot be used if the subquery contains an outer reference, as in this example:

List all of the offices whose targets exceed the sum of the quotas of the salespeople who
work in them:

```
  SELECT CITY
    FROM OFFICES
  WHERE TARGET > (SELECT SUM(QUOTA)
                   FROM SALESREPS
                  WHERE REP_OFFICE = OFFICE)

  CITY
  -----------
  Chicago
  Los Angeles
```

For each row of the OFFICES table to be tested by the WHERE clause of the main query, the OFFICE column (which appears in the subquery as an outer reference) has a different value. Thus SQL has no choice but to carry out this subquery five times—once for each row in the OFFICES table. A subquery containing an outer reference is called a correlated subquery because its results are correlated with each individual row of the main query. For the same reason, an outer reference is sometimes called a correlated reference.

A subquery can contain an outer reference to a table in the FROM clause of any query that contains the subquery, no matter how deeply the sub queries are nested. A column name in a fourth-level subquery, for example, may refer to one of the tables named in the FROM clause of the main query, or to a table named in the FROM clause of the second-level subquery or the third-level subquery that contains it. Regardless of the level of nesting, an outer reference always takes on the value of the column in the "current" row of the table being tested.

Because a subquery can contain outer references, there is even more potential for ambiguous column names in a subquery than in a main query. When an unqualified column name appears within a subquery, SQL must determine whether it refers to a table in the sub query's own FROM clause, or to a FROM clause in a query containing the subquery. To minimize the possibility of confusion, SQL always interprets a column reference in a subquery using the nearest FROM clause possible. To illustrate this point, in this example the same table is used in the query and in the subquery:

List the salespeople who are over 40 and who manage a salesperson over quota.

```
SELECT NAME
  FROM SALESREPS
WHERE AGE > 40
   AND EMPL_NUM IN (SELECT MANAGER
                    FROM SALESREPS
                   WHERE SALES > QUOTA)

NAME
-----------
Sam Clark
Larry Fitch
```

The MANAGER, QUOTA, and SALES columns in the subquery are references to the SALESREPS table in the sub query's own FROM clause; SQL does not interpret them as outer references, and the subquery is not a correlated subquery. As discussed earlier, SQL can perform the subquery first in this case, finding the salespeople who are over quota and generating a list of the employee numbers of their managers. SQL can then turn its attention to the main query, selecting managers whose employee numbers appear in the generated list.

If you want to use an outer reference within a subquery like the one in the previous example, you must use a table alias to force the outer reference. This request, which adds one more qualifying condition to the previous one, shows how:

List the managers who are over 40 and who manage a salesperson who is over quota and who does not work in the same sales office as the manager.

```
SELECT NAME
  FROM SALESREPS MGRS
WHERE AGE > 40
  AND MGRS.EMPL_NUM IN (SELECT MANAGER
                        FROM SALESREPS EMPS
                       WHERE EMPS.QUOTA > EMPS.SALES

                       AND EMPS.REP_OFFICE <>
MGRS.REP_OFFICE)

NAME
-----------
Sam Clark
Larry Fitch
```

The copy of the SALESREPS table used in the main query now has the tag MGRS, and the copy in the subquery has the tag EMPS. The subquery contains one additional search condition, requiring that the employee's office number does not match that of the manager. The qualified column name MGRS.OFFICE in the subquery is an outer reference, and this subquery is a correlated subquery.

4.10 EMBEDDED SQL

The central idea of embedded SQL is to blend SQL language statements directly into a program written in a "host" programming language, such as C, Pascal, COBOL, FORTRAN, PL/I, or Assembler. Embedded SQL uses the following techniques to embed the SQL statements:

• SQL statements are intermixed with statements of the host language in the source program. This "embedded SQL source program" is submitted to a SQL pre-compiler, which processes the SQL statements.

• Variables of the host programming language can be referenced in the embedded SQL statements, allowing values calculated by the program to be used by the SQL statements.

• Program language variables also are used by the embedded SQL statements to receive the results of SQL queries, allowing the program to use and process the retrieved values.

• Special program variables are used to assign NULL values to database columns and to support the retrieval of NULL values from the database.

• Several new SQL statements that are unique to embedded SQL are added to the interactive SQL language, to provide for row-by-row processing of query results.

4.10.1 Developing an Embedded SQL Program

An embedded SQL program contains a mix of SQL and programming language statements, so it can't be submitted directly to a compiler for the programming language. Instead, it moves through a multi-step development process, shown in the following Figure. The steps in the figure are actually those used by the IBM mainframe databases (DB2, SQL/DS), but all products that support embedded SQL use a similar process:

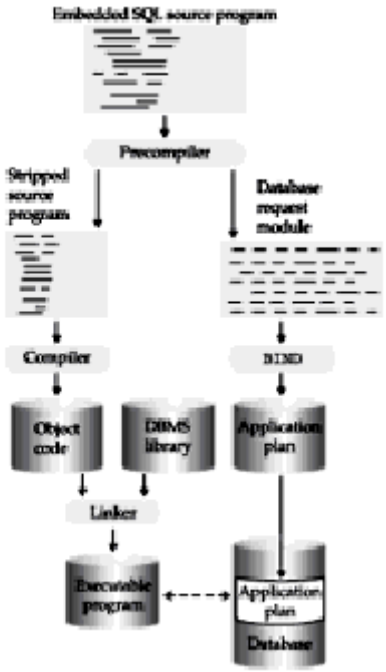


Figure: The embedded SQL development process

1. The embedded SQL source program is submitted to the SQL pre compiler, a programming tool. The pre compiler scans the program, finds the embedded SQL statements, and processes them. A different pre compiler is required for each programming language supported by the DBMS. Commercial SQL products typically offer pre compilers for one or more languages, including C, Pascal, COBOL, FORTRAN, Ada, PL/I, RPG, and various assembly languages.

2. The pre compiler produces two files as its output. The first file is the source program, stripped of its embedded SQL statements. In their place, the pre compiler substitutes calls to the "private" DBMS routines that provide the run-time link between the program and the DBMS. Typically, the names and calling sequences of these

routines are known only to the pre compiler and the DBMS; they are not a public interface to the DBMS. The second file is a copy of all the embedded SQL statements used in the program. This file is sometimes called a database request module, or DBRM.

3. The source file output from the pre compiler is submitted to the standard compiler for the host programming language (such as a C or COBOL compiler). The compiler processes the source code and produces object code as its output. Note that this step has nothing in particular to do with the DBMS or with SQL.

4. The linker accepts the object modules generated by the compiler, links them with various library routines, and produces an executable program. The library routines linked into the executable program include the "private" DBMS routines described in Step 2.

5. The database request module generated by the pre compiler is submitted to a special BIND program. This program examines the SQL statements, parses, validates, and optimizes them, and produces an application plan for each statement. The result is a combined application plan for the entire program, representing a DBMS-executable version of its embedded SQL statements. The BIND program stores the plan in the database, usually assigning it the name of the application program that created it.

The program development steps in Figure correlate with the DBMS statement processing steps in Figure. In particular, the pre compiler usually handles statement parsing (the first step), and the BIND utility handles verification, optimization, and plan generation (the second, third, and fourth steps). Thus the first four steps of Figure, all take place at compile time when you use embedded SQL. Only the fifth step, the actual execution of the application plan, remains to be done at run-time.

The embedded SQL development process turns the original embedded SQL source program into two executable parts:

• An executable program, stored in a file on the computer in the same format as any executable program

• An executable application plan, stored within the database in the format expected by the DBMS the embedded SQL development cycle may seem cumbersome, and it is more awkward than developing a standard C or COBOL program. In most cases, all of the steps in Figure are automated by a single command procedure, so the individual steps are made invisible to the application programmer. The process does have several major advantages from a DBMS point of view:

• The blending of SQL and programming language statements in the embedded SQL source program is an effective way to merge the two languages. The host programming language provides flow of control, variables, block structure, and input/output functions; SQL handles database access and does not have to provide these other constructs.

• The use of a pre compiler means that the compute-intensive work of parsing and optimization can take place during the development cycle. The resulting executable program is very efficient in its use of CPU resources.

• The database request module produced by the pre compiler provides portability of applications. An application program can be written and tested on one system, and then its executable program and DBRM can be moved to another system. After the BIND program on the new system creates the application plan and installs it in the database, the application program can use it without being recompiled itself.

• The program's actual run-time interface to the private DBMS routines is completely hidden from the application programmer. The programmer works with embedded SQL at the source code level and does not have to worry about other, more complex interfaces.

4.11 DYNAMIC SQL

The central concept of dynamic SQL is simple: don't hard-code an embedded SQL statement into the program's source code. Instead, let the program build the text of a SQL statement in one of its data areas at runtime. The program then passes the statement text to the DBMS for execution "on the fly." Although the details get quite complex, all of dynamic SQL is built on this simple concept, and it's a good idea to keep it in mind.

As you might expect, dynamic SQL is less efficient than static SQL. For this reason, static SQL is used whenever possible, and many application programmers never need to learn about dynamic SQL. However, dynamic SQL has grown in importance as more and more database access has moved to a client/server, front-end/back-end architecture over the last ten years. Database access from within personal computer applications such as spreadsheets and word processors has grown dramatically, and an entire set of PC-based front-end data entry and data access tools has emerged. All of these applications require the features of dynamic SQL.

More recently, the emergence of Internet-based "three-tier" architectures, with applications logic executing on one ("mid-tier")

system and the database logic executing on another ("back-end" system), have added new importance to capabilities that have grown out of dynamic SQL. In most of these three-tier environments, the applications logic running in the middle tier is quite dynamic. It must be changed frequently to respond to new business conditions and to implement new business rules. This frequently changing environment is at odds with the very tight coupling of applications programs and database contents implied by static SQL. As a result, most three-tier architectures use a callable SQL API (described in the next chapter) to link the middle tier to back-end databases. These APIs explicitly borrow the key concepts of dynamic SQL (for example, separate PREPARE and EXECUTE steps and the EXECUTE IMMEDIATE capability) to provide their database access. A solid understanding of dynamic SQL concepts is thus important to help a programmer understand what's going on "behind the scenes" of the SQL API. In performance-sensitive applications, this understanding can make all the difference between an application design that provides good performance and response times and one that does not.

4.12 TRIGGERS

The concept of a trigger is relatively straightforward. For any event that causes a change in the contents of a table, a user can specify an associated action that the DBMS should carry out. The three events that can trigger an action are attempts to INSERT, DELETE, or UPDATE rows of the table. The action triggered by an event is specified by a sequence of SQL statements.

To understand how a trigger works, let's examine a concrete example. When a new order is added to the ORDERS table, these two changes to the database should also take place:

• The SALES column for the salesperson who took the order should be increased by the amount of the order.

• The QTY_ON_HAND amount for the product being ordered should be decreased by the quantity ordered.

This Transact-SQL statement defines a SQL Server trigger, named NEWORDER that causes these database updates to happen automatically:

```
CREATE TRIGGER NEWORDER
 ON ORDERS
FOR INSERT
 AS UPDATE SALESREPS
      SET SALES = SALES + INSERTED.AMOUNT
     FROM SALESREPS, INSERTED
    WHERE SALESREPS.EMPL_NUM = INSERTED.REP
   UPDATE PRODUCTS
```

```
 SET QTY_ON_HAND = QTY_ON_HAND - INSERTED.QTY
FROM PRODUCTS, INSERTED
WHERE PRODUCTS.MFR_ID = INSERTED.MFR
 AND PRODUCTS.PRODUCT_ID = INSERTED.PRODUCT
```

The first part of the trigger definition tells SQL Server that the trigger is to be invoked whenever an INSERT statement is attempted on the ORDERS table. The remainder of the definition (after the keyword AS) defines the action of the trigger. In this case, the action is a sequence of two UPDATE statements, one for the SALESREPS table and one for the PRODUCTS table. The row being inserted is referred to using the pseudo-table name inserted within the UPDATE statements. As the example shows, SQL Server extends the SQL language substantially to support triggers. Other extensions not shown here include IF/THEN/ELSE tests, looping, procedure calls, and even PRINT statements that display user messages.

The trigger capability, while popular in many DBMS products, is not a part of the ANSI/ISO SQL2 standard. As with other SQL features whose popularity has preceded standardization, this has led to a considerable divergence in trigger support across various DBMS brands. Some of the differences between brands are merely differences in syntax. Others reflect real differences in the underlying capability.

DB2's trigger support provides an instructive example of the differences. Here is the same trigger definition shown previously for SQL Server, this time using the DB2 syntax:

```
CREATE TRIGGER NEWORDER
    AFTER INSERT ON ORDERS
REFERENCING NEW AS NEW_ORD
  FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
    UPDATE SALESREPS
      SET SALES = SALES + NEW_ORD.AMOUNT
     WHERE SALESREPS.EMPL_NUM = NEW_ORD.REP;
    UPDATE PRODUCTS
      SET QTY_ON_HAND = QTY_ON_HAND –
NEW_ORD.QTY
     WHERE PRODUCTS.MFR_ID = NEW_ORD.MFR
      AND PRODUCTS.PRODUCT_ID =
NEW_ORD.PRODUCT;
      END
```

The beginning of the trigger definition includes the same elements as the SQL Server definition, but rearranges them. It explicitly tells DB2 that the trigger is to be invoked AFTER a new order is inserted into the database. DB2 also allows you to specify

that the trigger is to be carried out before a triggering action is applied to the database contents. This doesn't make sense in this example, because the triggering event is an INSERT operation, but it does make sense for UPDATE or DELETE operations.

The DB2 REFERENCING clause specifies a table alias (NEW_ORD) that will be used to refer to the row being inserted throughout the remainder of the trigger definition. It serves the same function as the INSERTED keyword in the SQL Server trigger. The statement references the "new" values in the inserted row because this is an INSERT operation trigger. For a DELETE operation trigger, the "old" values would be referenced. For an UPDATE operation trigger, DB2 gives you the ability to refer to both the "old" (pre-UPDATE) values and "new" (post-UPDATE) values.

The BEGIN ATOMIC and END serve as brackets around the sequence of SQL statements that define the triggered action. The two searched UPDATE statements in the body of the trigger definition are straightforward modifications of their SQL Server counterparts. They follow the standard SQL syntax for searched UPDATE statements, using the table alias specified by the REFERENCING clause to identify the particular row of the SALESREPS table and the PRODUCTS table to be updated. The row being inserted is referred to using the pseudo-table name inserted within the UPDATE statements.

Here is another example of a trigger definition, this time using Informix Universal Server:

```
CREATE TRIGGER NEWORDER
      INSERT ON ORDERS
       AFTER (EXECUTE PROCEDURE NEW_ORDER)
```

This trigger again specifies an action that is to take place AFTER a new order is inserted. In this case, the multiple SQL statements that form the triggered action can't be specified directly in the trigger definition. Instead, the triggered statements are placed into an Informix stored procedure, named NEW_ORDER and the trigger causes the stored procedure to be executed. As this and the preceding examples show, although the core concepts of a trigger mechanism are very consistent across databases, the specifics vary a great deal. Triggers are certainly among the least portable aspects of SQL databases today.

4.12.1Trigger Advantages and Disadvantages

A complete discussion of triggers is beyond the scope of this book, but even these simple examples shows the power of the trigger mechanism. The major advantage of triggers is that business rules can be stored in the database and enforced

consistently with each update to the database. This can dramatically reduce the complexity of application programs that access the database. Triggers also have some disadvantages, including these:

• Database complexity. When the rules are moved into the database, setting up the database becomes a more complex task. Users who could reasonably be expected to create small, ad hoc applications with SQL will find that the programming logic of triggers makes the task much more difficult.

• Hidden rules. With the rules hidden away inside the database, programs that appear to perform straightforward database updates may, in fact, generate an enormous amount of database activity. The programmer no longer has total control over what happens to the database. Instead, a program-initiated database action may cause other, hidden actions.

• Hidden performance implications. With triggers stored inside the database, the consequences of executing a SQL statement are no longer completely visible to the programmer. In particular, an apparently simple SQL statement could, in concept, trigger a process that involves a sequential scan of a very large database table, which would take a long time to complete. These performance implications of any given SQL statement are invisible to the programmer.

5

DATABASE APPLICATION DEVELOPMENT

Topics covered

5.1 Accessing Databases From Applications

5.2 Cursors

5.3 JDBC Driver Management

5.4 Executing SQL Statements

5.5 ResultSets

5.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we cover how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called Embedded SQL. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

5.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straight-forward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL. In particular, some special host language variables must be declared in SQL (so that, for example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language).

There are, however, two complications to bear in mind. First, the data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by casting data values appropriately before passing them to or from SQL commands. (SQL, like other programming languages, provides an operator to cast values of alle type into values of another type.) The second complication has to do with SQL being set-oriented, and is addressed using cursors

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION. The declarations are similar to how they would look in a C program and, as usual in C. are separated by semicolons. For example. we can declare variables c-sname, c_sid, c_rating, and c_age (with the initial c used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, c_snamc has the type CHARACTER (20) when referred to in an SQL statement, csid has the type INTEGER, crating has the type SMALLINT, and cage has the type REAL. We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE. SQLCODE is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes. SQLSTATE, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported. One of these two variables must be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char [6J , that is, a character string five characters long.

5.2 CURSORS

A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on set" of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation. This mechanism is called a cursor. We can declare a cursor on any relation or on any SQL query (because

every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next n, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

5.2.1 Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of JWS computed by an Embedded SQL statement:

1) We usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.

2) NSERT, DELETE, and UPDATE statements typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable c_sid, declared earlier, as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE   S.sid = :c_sid;
```

This query returns a collection of rows, not just one row. When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the command with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor sinfo is defined. Subsequently, we can open the cursor:

OPEN sinfo:

The value of c_minrating in the SQL query associated with the cursor is the value of this variable when we open the cursor.

(The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor sinfo into host language variables:

FETCH sinfo INTO: csname, cage;

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways.

5.3 JDBC DRIVER MANAGEMENT

In .Jdbc, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers. The Drivermanager class has methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished by using the Java mechanism for dynamically loading classes. The static method forName in the Class class returns the Java class as specified in the argument string and executes its static constructor.

The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

The following Java example code explicitly loads a JDBC driver:

Class.forName("oracle/jdbc.driver.OracleDriver");

There are two other ways ofregistering a driver. We can include the driver with jdbc. drivers=oracle/jdbc.driver at the command line when we start the Java application. Alternatively, we can explicitly instantiate a driver, but this method is used only rarely, as the name of the driver has to be specified in the application code, and thus the application becomes sensitive to

changes at the driver level. After registering the driver, we connect to the data source.

5.3.1 Connections

A session with a data source is started through creation of a Connection object; A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source. Connections are specified through a JDBC URL, a URL that uses the jdbc protocol. Such a URL has the form
jdbc:<subprotocol>:<otherParameters>

The following code example establishes a connection to an Oracle database assuming that the strings userId and password are set to valid values. In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If autocommit is set for a connection, then each SQL statement is considered to be its own transaction. If autocommit is off, then a series of statements that compose a transaction can be committed using the commit() method of the Connection class, or aborted using the rollback() method. The Connection class has methods to set the

```
String uri = “jdbc:oracle:www.bookstore.com:3083”
Connection connection;
try {
Connection connection =
DriverManager.getConnection(urI,userId,password);
}
catch(SQLException excpt) {
System.out.println(excpt.getMessageO);
return;
}
```

autocommit mode (Connection. setAutoCommi t) and to retrieve the current autocommit mode (getAutoCommit). The following methods are part of the. Connection interface and permit setting and getting other properties:

• public int getTransactionIsolation() throws SQLExceptionand
public void setTransactionIsolation(int 1) throws SQLException.
These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation (are possible, and argument 1 can be
set as follows:
- TRANSACTIONJNONE
- TRANSACTIONJREAD.UNCOMMITTED
- TRANSACTIONJREAD.COMMITTED
- TRANSACTIONJREPEATABLEJREAD
- TRANSACTION.BERIALIZABLE

• public boolean getReadOnlyO throws SQLException and public void setReadOnly(boolean readOnly) throws SQLException. These two functions allow the user to specify whether the transactions executecl through this connection are read only.

⇨ public boolean isClosed() throws SQLException.
Checks whether the current connection has already been closed.
⇨ setAutoCommi t and get AutoCommi t.

We already discussed these two functions. Establishing a connection to a data source is a costly operation since it involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory. In case an application establishes many different connections from different parties (such as a Web server), connections are often pooled to avoid this overhead. A con- nection pool is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source. Connection pooling can be handled either by specialized code in the application, or the optional j avax. sql package, which provides functionality for connection pooling and allows us to set different parameters, such as the capacity of the pool, and shrinkage and growth rates.

5.4 EXECUTING SQL STATEMENTS

We now discuss how to create and execute SQL statements using JDBC. In the JDBC code examples in this section, we assume that we have a Connection object named con. JDBC supports three different ways of executing statements: Statement, PreparedStatement, and CallableStatement. The Statement class is the base class for the other two statment classes. It allows us to query the data source with any static or dynamically generated SQL query. The PreparedStatement cla,Cis dynamically generates precompiled SQL state- ments that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object (representing the SQL statement) is created.

Consider the sample code using a PreparedStatment object shown in following Figure The SQL query specifies the query string, but uses "1' for the values of the parameters, which are set later using methods setString, setFloat, and setlnt. The "1' placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements.

```
// initial quantity is always zero
String sql = "INSERT INTO Books VALUES(?, 7, ?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);

// now instantiate the parameters with values
// assume that isbn, title, etc. are Java variables that
// contain the values to be inserted
pstmt.clearParameters();
pstmt.setString(1, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);

int numRows = pstmt.executeUpdate();
```

The method setString is one way to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters 0 before setting parameter values in order to remove any old data. There are different ways of submitting the query string to the data source. In the example, we used the executeUpdate command, which is used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The executeUpdate method returns an integer indicating the number of rows the SQL statement modified; it returns 0 for successful execution without modifying any rows. The executeQuery method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object, which we discuss next.

5.5 RESULTSETS

As discussed in the previous section, the statement executeQuery returns a ResultSet object, which is similar to a cursor. ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions.

In its most basic form, the ResultSet object allows us to read one row of the output of the query at a time. Initially, the ResultSet is positioned before the first row, and we have to retrieve the first row with an explicit call to the next() method. The next method returns false if there are no more rows in the query answer, and

true other\vise. The code fragment shown in following Figure illustrates the basic usage of a ResultSet object.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
/ / rs is now a cursor
/ / first call to rs.nextO moves to the first record
/ / rs.nextO moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
/ / process the data
}
```

Figure: Using a ResultSet Object

While next () allows us to retrieve the logically next row in the query answer, we can move about in the query answer in other ways too:

• previous() moves back one row.

• absolute (int num) moves to the row with the specified number.

• relative (int num) moves forward or backward (if num is negative) relative to the current position. relative (-1) has the same effect as previous.

• first 0 moves to the first row, and last 0 moves to the last row.

6

OVERVIEW OF STORAGE AND INDEXING

Topics covered

6.1 Storage Hierarchies

6.2 Tree structured indexing and Hash Based indexing

6.1 STORAGE HIERARCHIES

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

• **Primary storage**: This category includes storage media that can be operated on directly by the computer central processing unit (CPU), such as the computer main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity.

• **Secondary storage**: This category includes magnetic disks, optical disks, and tapes. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary storage cannot be processed directly by the CPU; it must first be copied into primary storage.

6.1.1 Memory Hierarchies and Storage Devices

In a modern computer system data resides and is transported throughout a hierarchy of storage media. The highest-speed memory is the most expensive and is therefore available with the least capacity. The lowest-speed memory is tape storage, which is essentially available in indefinite storage capacity. At the primary storage level, the memory hierarchy includes at the most expensive end cache memory, which is a static RAM (Random Access Memory). Cache memory is typically used by the CPU to speed up execution of programs. The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping programs and data and is popularly called main memory. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility and lower speed compared with static RAM. At the secondary storage level,

the hierarchy includes magnetic disks, as well as mass storage in the form of CD-ROM (Compact Disk–Read-Only Memory) devices, and finally tapes at the least expensive end of the hierarchy. The storage capacity is measured in kilobytes (Kbyte or 1000 bytes), megabytes (Mbyte or 1 million bytes), gigabytes (Gbyte or 1 billion bytes), and even terabytes (1000 Gbytes). Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, and portions of the database are read into and written from buffers in main memory as needed.

Now that personal computers and workstations have tens of megabytes of data in DRAM, it is becoming possible to load a large fraction of the database into main memory. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), leading to main memory databases; these are particularly useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.

Between DRAM and magnetic disk storage, another form of memory, flash memory, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over at a time.

CD-ROM disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten. WORM (Write-Once-Read-Many) disks are a form of optical storage used for archiving data; they allow data to be written once and read any number of times without the possibility of erasing. They hold about half a gigabyte of data per disk and last much longer than magnetic disks. Optical juke box memories use an array of CD-ROM platters, which are loaded onto drives on demand. Although optical juke boxes have capacities in the hundreds of gigabytes, their retrieval times are in the hundreds of milliseconds, quite a bit slower than magnetic disks. This type of storage has not become as popular as it was expected to be because of the rapid decrease in cost and increase in capacities of magnetic disks. The DVD (Digital Video Disk) is a recent standard for optical disks allowing four to fifteen gigabytes of storage per disk. Finally, magnetic tapes are used for archiving and backup storage of data. Tape jukeboxes—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as tertiary storage to hold terabytes of data.

For example, NASA's EOS (Earth Observation Satellite) system stores archived databases in this fashion. It is anticipated that many large organizations will find it normal to have terabyte sized databases in a few years. The term very large database cannot be defined precisely anymore because disk storage capacities are on the rise and costs are declining. It may very soon be reserved for databases containing tens of terabytes.

6.1.2 Storage of Databases

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

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

• The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as nonvolatile storage, whereas main memory is often called volatile storage.

• The cost of storage per unit of data is an order of magnitude less for disk than for primary storage.

Some of the newer technologies—such as optical disks, DVDs, and tape jukeboxes—are likely to provide viable alternatives to the use of magnetic disks. Databases in the future may therefore reside at different levels of the memory hierarchy. For now, however, it is important to study and understand the properties and characteristics of magnetic disks and the way data files can be organized on disk in order to design effective databases with acceptable performance.

Magnetic tapes are frequently used as a storage medium for backing up the database because storage on tape costs even less than storage on disk. However, access to data on tape is quite slow. Data stored on tapes is off-line; that is, some intervention by an operator—or an automatic loading device—to load a tape is needed before this data becomes available. In contrast, disks are on-line devices that can be accessed directly at any time.

The techniques used to store large amounts of structured data on disk are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific

DBMS. Usually, the DBMS has several options available for organizing the data, and the process of physical database design involves choosing from among the options the particular data organization techniques that best suit the given application requirements. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as files of records. Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently whenever they are needed.

There are several primary file organizations, which determine how the records of a file are physically placed on the disk, and hence how the records can be accessed. A heap file (or unordered file) places the records on disk in no particular order by appending new records at the end of the file, whereas a sorted file (or sequential file) keeps the records ordered by the value of a particular field (called the sort key). A hashed file uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk. Other primary file organizations, such as B-trees, use tree structures. A secondary organization or auxiliary access structure allows efficient access to the records of a file based on alternate fields than those that have been used for the primary file organization.

6.2 TREE STRUCTURED INDEXING AND HASH BASED INDEXING

- Hash-based indexes are best for *equality selections*. ***Cannot*** support range searches.
- B^+-trees are best for sorted access and range queries.

6.2.1 Tree Structured Indexing

The data entries are arranged in sorted order by search key value, and a hierarchical search data structure is maintained that directs searches to the correct page of data entries.
Following Figure 8.3 shows the employee records, organized in a tree-structured index with search key age. Each node in this figure (e.g., nodes labeled A, B, L1, L2) is a physical page, and retrieving a node involves a disk I/O.

The lowest level of the tree, called the leaf level, contains the data entries; in our example, these are employee records. To illustrate the ideas better, we have drawn the Figure as if there were additional employee records, some with age less than 22 and some with age greater than EiO. Additional records with age less than 22 would appear in leaf pages to the left page L1 and records with age greater than 50 would appear in leaf pages to the right of page L3.

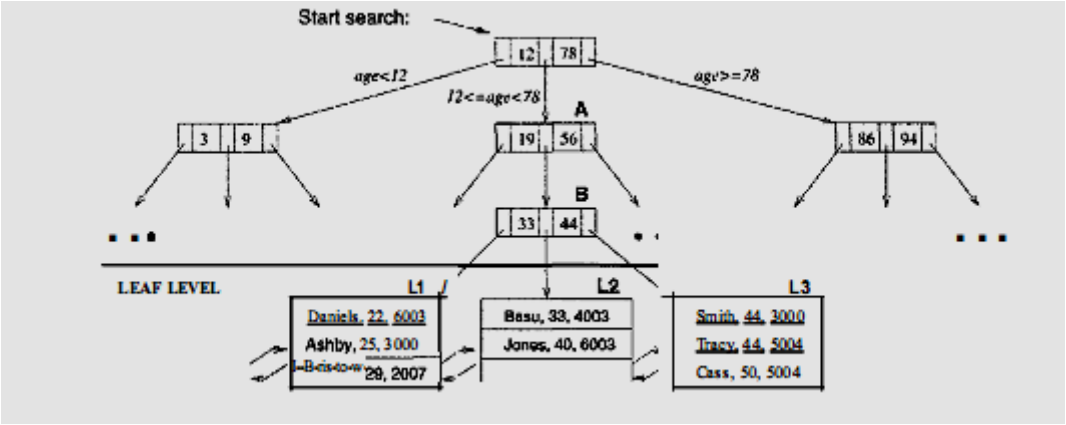


Figure: Tree-Structured Index

This structure allows us to efficiently locate all data entries with search key values in a desired range. All searches begin at the topmost node, called the root, and the contents of pages in non-leaf levels direct searches to the correct leaf page. Non-leaf pages contain node pointers separated by search key values. The node pointer to the left of a key value k points to a subtree that contains only data entries less than k. The node pointer to the right of a key value k points to a subtree that contains only data entries greater than or equal to k.

In our example, suppose we want to find all data entries with 24 < age < 50.

In our example search, we look for data entries with search key value > 24, and get directed to the middle child, node A. Again, examining the contents of this node, we are directed to node B. Examining the contents of node B, we are directed to leaf node LI, which contains data entries we are looking for. Observe that leaf nodes L2 and L3 also contain data entries that satisfy our search criterion. To facilitate retrieval of such qualifying entries during search, all leaf pages are maintained in a doubly-linked list. Thus, we can fetch page L2 using the 'next' pointer on page LI, and then fetch page L3 using the 'next' pointer on L2. Thus, the number of disk I/Os incurred during a search is equal to the length of a path from the root to a leaf, plus the number of leaf pages with qualifying data entries. The B+ tree is an index structure that ensures that all paths from the root to a leaf in a given tree are of the same length,

that is, the structure is always balanced in height. Finding the correct leaf page is faster than binary search of the pages in a sorted file because each non-leaf node can accommodate a very large number of node-pointers, and the height of the tree is rarely more than three or four in practice. The height of a balanced tree is the length of a path from root to leaf; in Figure 8.3, the height is three. The number of l/Os to retrieve a desired leaf page is four, including the root and the leaf page. (In practice, the root is typically in the buffer pool because it is frequently accessed, and we really incur just three l/Os for a tree of height three.)

The average number of children for a non-leaf node is called the fan-out of the tree. If every non-leaf node has n children, a tree of height h has n^h leaf pages. In practice, nodes do not have the same number of children, but using the average value F for n, we still get a good approximation to the number of leaf pages, F^h . In practice, F is at least 100, which means a tree of height four contains 100 million leaf pages. Thus, we can search a file with 100 million leaf pages and find the page we want using four l/Os; in contrast, binary search of the same file would take log21OO,000,000 (over 25) l/Os.

6.2.2 Hash Based indexing

We can organize records using a technique called hashing to quickly find records that have a given search key value. For example, if the file of employee records is hashed on the name field, we can retrieve all records about Joe. In this approach, the records in a file are grouped in buckets, where a bucket consists of a primary page and, possibly, additional pages linked in a chain. The bucket to which a record belongs can be determined by applying a special function, called a hash function, to the search key. Given a bucket number, a hash-based index structure allows us to retrieve the primary page for the bucket in one or two disk l/Os. On inserts, the record is inserted into the appropriate bucket, with 'overflow' pages allocated as necessary. To search for a record with a given search key value, we apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket. If we do not have the search key value for the record, for example, the index is based on sal and we want records with a given age value, we have to scan all pages in the file. In this chapter, we assume that applying the hash function to (the search key of) a record allows us to identify and retrieve the page containing the record with one l/O. In practice, hash-based index structures that adjust gracefully to inserts and deletes and allow us to retrieve the page containing a record in one to two l/Os are known. Hash indexing is illustrated in Figure below, where the data is stored in a file that is hashed on age; the data entries in this first index file are the actual data records. Applying the hash function to the age field identifies the page that the record belongs to. The

hash function h for this example is quite simple; it converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier. The Figure also shows an index with search key sal that contains (sal, rid) pairs as data entries. The tid (short for record id) component of a data entry in this second index is a pointer to a record with search key value sal (and is shown in the figure as an arrow pointing to the data record). The file of employee records is hashed on age, and Alternative (1) is used for for data entries. The second index, on sal, also uses hashing to locate data entries, which are now (sal, rid of employee recoT'(~ pairs; that is, Alternative (2) is used for data entries.

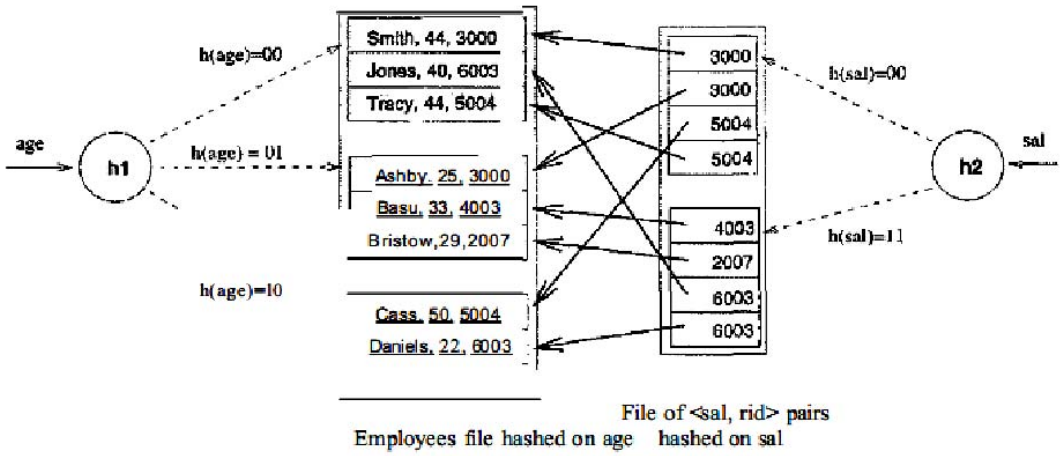


Figure: Index-Organized File Hashed on age, with Auxiliary Index on sal

Note that the search key for an index can be any sequence of one or more fields, and it need not uniquely identify records. For example, in the salary index, two data entries have the same search key value 6003.

Summary:

- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search.
  - Files rarely kept sorted in practice; B+ tree index is better.
- Index is a collection of data entries plus a way to quickly find entries with given key values.

- Data entries can be :
  - actual data records,
  - <key, rid> pairs, or
  - <key, rid-list> pairs.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered,
- Differences have important consequences for utility/performance of query processing
- Several kinds of integrity constraints can be expressed in the ER model: *key constraints*, *participation constraints*, and *overlap /covering constraints* for ISA hierarchies. Some *foreign key constraints* are also implicit in the definition of a relationship set.
  - Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model.
  - Constraints play an important role in determining the best database design for an enterprise.

7

QUERY EVALUATION OVERVIEW

Topics covered
7.1 Overview of Query optimization
7.2 Relational optimization

7.1 OVERVIEW OF QUERY OPTIMIZATION

Query optimization is a function of many relational database management systems in which multiple query plans for satisfying a query are examined and a good query plan is identified. This may or not be the absolute best strategy because there are many ways of doing plans. There is a trade-off between the amount of time spent figuring out the best plan and the amount running the plan. Different qualities of database management systems have different ways of balancing these two. Cost based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection.

Typically the resources which are costed are CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism. The set of query plans examined is formed by examining possible access paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join, product join). The search space can become quite large depending on the complexity of the SQL query. There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query. In addition there is physical optimization which is used to determine the means of carrying out each operation.

7.1.1 Query Evaluation Plan

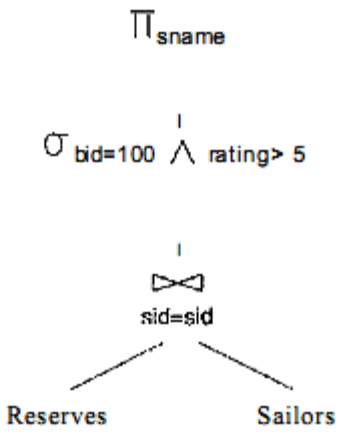
A query evaluation plan (or simply plan) consists of an extended relational algebra tree, with additional annotations at each node indicating the access methods to use for each table and the implementation method to use for each relational operator.
Consider the following SQL query:

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
```

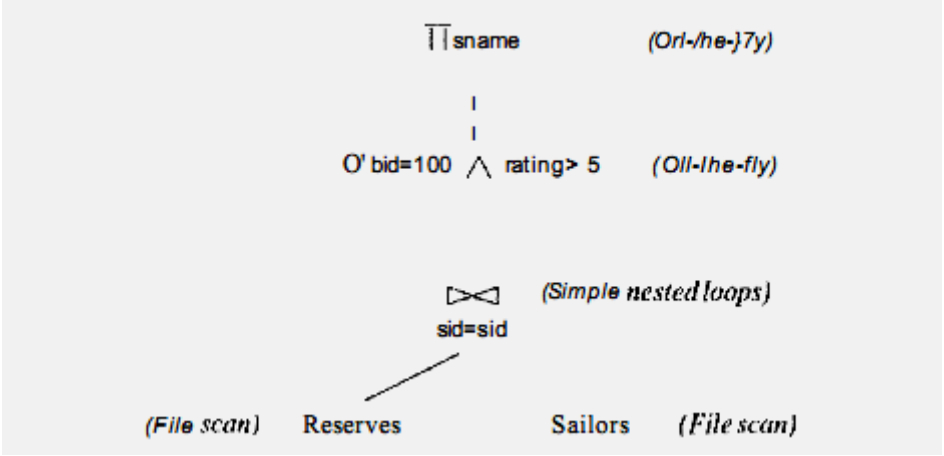
This query can be expressed in relational algebra as follows:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

This expression is shown in the form of a tree in following Figure. The algebra expression partially specifies how to evaluate the query-owe first compute the natural join of Reserves and Sailors, then perform the selections, and finally project the snarne field.



To obtain a fully specified evaluation plan, we must decide on an implementation for each of the algebra operations involved. }or example, we can use a page-oriented simple nested loops join with Reserves as the outer table and apply selections and projections to each tuple in the result of the join as it is produced; the result of the join before the selections and projections is never stored in its entirety.



In drawing the query evaluation plan, we have used the convention that the outer table is the left child of the join operator.

7.2 RELATIONAL OPTIMIZATION: - cost of a plan estimating result sizes

For each enumerated plan, we have to estimate its cost. There are two parts to estimating the cost of an evaluation plan for a query block:

1. For each node in the tree, we must estimate the cost of performing the corresponding operation. Costs are affected significantly by whether pipelining is used or temporary relations are created to pass the output of an operator to its parent.

2. For each node in the tree, we must estimate the size of the result and whether it is sorted. This result is the input for the operation that corresponds to the parent of the current node, and the size and sort order in turn affect the estimation of size, cost, and sort order for the panmt. As we saw there, estimating costs requires knowledge of various parameters of the input relations, such as the number of pages and available indexes. Such statistics are maintained in the DBMS's system catalogs. In this section, we describe the statistics maintained by a typical DBMS and discuss how result sizes are estimated. We use the number of page I/Os as the metric of cost and ignore issues such as blocked access, for the sake of simplicity. The estimates used by a DBMS for result sizes and costs are at best approximations to actual sizes and costs. It is unrealistic to expect an optimizer to find the very best plan; it is more important to avoid the worst plans and find a good plan.

7.2.1 Estimating Result Sizes

We now discuss how a typical optimizer estimates the size of the result computed by an operator on given inputs. Size estimation plays an important role in cost estimation as well because the output of one operator can be the input to another operator, and the cost of an operator depends on the size of its inputs.

Consider a query block of the form:

```
SELECT  attTibute list
FROM    Telation list
WHERE   teTml 1\ teTm2 1\ ... 1\ teTm_n
```

The maximum number of tuples in the result of this query (without duplicate elimination) is the product of the cardinalities of

the relations in the FROM clause. Every term in the WHERE clause, however, eliminates some of these potential result tuples. We can model the effect of the WHERE clause on the result size by associating a reduction factor with each term, which is the ratio of the (expected) result size to the input size considering only the selection represented by the term. The actual size of the result can be estimated as the maximum size times the product of the reduction factors for the terms in the WHERE clause. Of course, this estimate reflects the unrealistic but simplifying assumption that the conditions tested by each term are statistically independent.

8

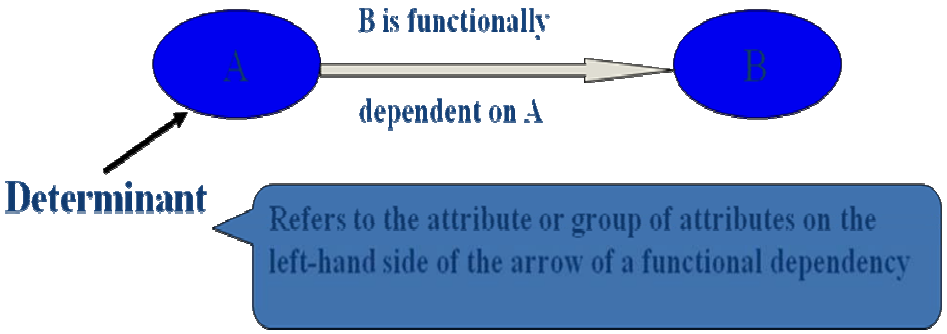
SCHEMA REFNEENT AND NORMAL FORMS

Topics covered

8.1 Functional Dependencies
8.2 Second Normal Form
8.3 Third Normal Form
8.4 Fourth Normal Form
8.6 Fifth Normal Form
8.7 BCNF
8.8 Comparison of 3NF and BCNF
8.9 Lossless and dependency preserving decomposition
8.10 Closure of Dependencies
8.11 Minimal Closure (Cover)

8.1 FUNCTIONAL DEPENDENCIES:

Functional dependency describes the relationship between attributes in a relation. For example, if A and B are attributes of relation R, and B is functionally dependent on A (denoted A B), if each value of A is associated with exactly one value of B. (A and B may each consist of one or more attributes.)



Trival functional dependency means that the right-hand side is a subset (not necessarily a proper subset) of the left-hand side. They do not provide any additional information about possible integrity constraints on the values held by these attributes.

We are normally more interested in **nontrivial dependencies** because they represent integrity constraints for the relation.

Main characteristics of functional dependencies in normalization

- Have a one-to-one relationship between attribute(s) on the left- and right- hand side of a dependency;
- hold for all time;
- are nontrivial.

Functional dependency is a property of the meaning or semantics of the attributes in a relation. When a functional dependency is present, the dependency is specified as a **constraint** between the attributes.

An important integrity constraint to consider first is **the identification of candidate keys, one of which is selected to be the primary key** for the relation using functional dependency.

8.1 FIRST NORMAL FORM:

A basic objective of the first normal form defined by Codd in 1970 was to permit data to be queried and manipulated using a "universal data sub-language" grounded in first-order logic. SQL is an example of such a data sub-language, albeit one that Codd regarded as seriously flawed.). Querying and manipulating the data within an unnormalized data structure, such as the following non-1NF representation of customers' credit card transactions, involves more complexity than is really necessary:

Customer Transactions

	Tr. ID	**Date**	**Amount**
Jones	12890	14-Oct-2003	-87
	12904	15-Oct-2003	-50
Wilkins	**Tr. ID**	**Date**	**Amount**
	12898	14-Oct-2003	-21
Stevens	**Tr. ID**	**Date**	**Amount**
	12907	15-Oct-2003	-18
	14920	20-Nov-2003	-70
	15003	27-Nov-2003	-60

To each customer there corresponds a *repeating group* of transactions. The automated evaluation of any query relating to customers' transactions therefore would broadly involve two stages:

1. Unpacking one or more customers' groups of transactions allowing the individual transactions in a group to be examined, and
2. Deriving a query result based on the results of the first stage

For example, in order to find out the monetary sum of all transactions that occurred in October 2003 for all customers, the system would have to know that it must first unpack the *Transactions* group of each customer, then sum the *Amounts* of all transactions thus obtained where the *Date* of the transaction falls in October 2003.

One of Codd's important insights was that this structural complexity could always be removed completely, leading to much greater power and flexibility in the way queries could be formulated (by users and applications) and evaluated (by the DBMS). The normalized equivalent of the structure above would look like this:

Customer	Tr. ID	Date	Amount
Jones	12890	14-Oct-2003	-87
Jones	12904	15-Oct-2003	-50
Wilkins	12898	14-Oct-2003	-21
Stevens	12907	15-Oct-2003	-18
Stevens	14920	20-Nov-2003	-70
Stevens	15003	27-Nov-2003	-60

Now each row represents an individual credit card transaction, and the DBMS can obtain the answer of interest, simply by finding all rows with a Date falling in October, and summing their Amounts. All of the values in the data structure are on an equal footing: they are all exposed to the DBMS directly, and can directly participate in queries, whereas in the previous situation some values were embedded in lower-level structures that had to be handled specially. Accordingly, the normalized design lends itself to general-purpose query processing, whereas the unnormalized design does not.

The objectives of normalization beyond 1NF were stated as follows by Codd:

1. To free the collection of relations from undesirable insertion, update and deletion dependencies;
2. To reduce the need for restructuring the collection of relations as new types of data are introduced, and thus increase the life span of application programs;
3. To make the relational model more informative to users;

4. To make the collection of relations neutral to the query statistics, where these statistics are liable to change as time goes by.

—E.F. Codd, "Further Normalization of the Data Base Relational Model.

8.2 SECOND NORMAL FORM

Second normal form (**2NF**) is a normal form used in database normalization. 2NF was originally defined by E.F. Codd in 1971. A table that is in first normal form (1NF) must meet additional criteria if it is to qualify for second normal form. Specifically: a 1NF table is in 2NF if and only if, given any candidate key K and any attribute A that is not a constituent of a candidate key, A depends upon the whole of K rather than just a part of it.

In slightly more formal terms: a 1NF table is in 2NF if and only if all its non-prime attributes are functionally dependent on the whole of a candidate key. (A non-prime attribute is one that does not belong to any candidate key.)

Note that when a 1NF table has no composite candidate keys (candidate keys consisting of more than one attribute), the table is automatically in 2NF.

Consider a table describing employees' skills:

Employees' Skills

Employee	**Skill**	**Current Work Location**
Jones	Typing	114 Main Street
Jones	Shorthand	114 Main Street
Jones	Whittling	114 Main Street
Bravo	Light Cleaning	73 Industrial Way
Ellis	Alchemy	73 Industrial Way
Ellis	Flying	73 Industrial Way
Harrison	Light Cleaning	73 Industrial Way

Neither {Employee} nor {Skill} is a candidate key for the table. This is because a given Employee might need to appear more than once (he might have multiple Skills), and a given Skill might need to appear more than once (it might be possessed by multiple Employees). Only the composite key {Employee, Skill} qualifies as a candidate key for the table.

The remaining attribute, Current Work Location, is dependent on only part of the candidate key, namely Employee. Therefore the table is not in 2NF. Note the redundancy in the way

Current Work Locations are represented: we are told three times that Jones works at 114 Main Street, and twice that Ellis works at 73 Industrial Way. This redundancy makes the table vulnerable to update anomalies: it is, for example, possible to update Jones' work location on his "Typing" and "Shorthand" records and not update his "Whittling" record. The resulting data would imply contradictory answers to the question "What is Jones' current work location?"

A 2NF alternative to this design would represent the same information in two tables: an "Employees" table with candidate key {Employee}, and an "Employees' Skills" table with candidate key {Employee, Skill}:

Employees

Employee	**Current Work Location**
Jones	114 Main Street
Bravo	73 Industrial Way
Ellis	73 Industrial Way
Harrison	73 Industrial Way

Employees' Skills

Employee	**Skill**
Jones	Typing
Jones	Shorthand
Jones	Whittling
Bravo	Light Cleaning
Ellis	Alchemy
Ellis	Flying
Harrison	Light Cleaning

Neither of these tables can suffer from update anomalies.

Not all 2NF tables are free from update anomalies, however. An example of a 2NF table which suffers from update anomalies is:

Tournament Winners

Tournament	**Year**	**Winner**	**Winner Date of Birth**
Des Moines Masters	1998	Chip Masterson	14 March 1977
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Even though Winner and Winner Date of Birth are determined by the whole key {Tournament / Year} and not part of it, particular Winner / Winner Date of Birth combinations are shown redundantly on multiple records. This leads to an update anomaly: if updates are not carried out consistently, a particular winner could be shown as having two different dates of birth.

The underlying problem is the transitive dependency to which the Winner Date of Birth attribute is subject. Winner Date of Birth actually depends on Winner, which in turn depends on the key Tournament / Year.

This problem is addressed by third normal form (3NF).

8.3 THIRD NORMAL FORM

The **third normal form** (**3NF**) is a normal form used in database normalization. 3NF was originally defined by E.F. Codd in 1971. Codd's definition states that a table is in 3NF if and only if both of the following conditions hold:

- The relation R (table) is in second normal form (**2NF**)
- Every non-prime attribute of R is non-transitively dependent (i.e. directly dependent) on every key of R.
-

A **non-prime attribute** of R is an attribute that does not belong to any candidate key of R. A transitive dependency is a functional dependency in which $X \rightarrow Z$ (X determines Z) indirectly, by virtue of $X \rightarrow Y$ and $Y \rightarrow Z$ (where it is not the case that $Y \rightarrow X$).

A 3NF definition that is equivalent to Codd's, but expressed differently was given by Carlo Zaniolo in 1982. This definition states that a table is in 3NF if and only if, for each of its functional dependencies $X \rightarrow A$, **at least one** of the following conditions holds:

- X contains A (that is, $X \rightarrow A$ is trivial functional dependency), or
- X is a superkey, or
- A is a **prime attribute** (i.e., A is contained within a candidate key)
- n example of a 2NF table that fails to meet the requirements of 3NF is:

Tournament Winners

Tournament	Year Winner	Winner Date of Birth
Indiana Invitational	1998 Al Fredrickson	21 July 1975
Cleveland Open	1999 Bob Albertson	28 September 1968
Des Moines Masters	1999 Al Fredrickson	21 July 1975
Indiana Invitational	1999 Chip Masterson	14 March 1977

- Because each row in the table needs to tell us who won a particular Tournament in a particular Year, the composite key {Tournament, Year} is a minimal set of attributes guaranteed to uniquely identify a row. That is, {Tournament, Year} is a candidate key for the table.

- The breach of 3NF occurs because the non-prime attribute Winner Date of Birth is transitively dependent on the candidate key {Tournament, Year} via the non-prime attribute Winner. The fact that Winner Date of Birth is functionally dependent on Winner makes the table vulnerable to logical inconsistencies, as there is nothing to stop the same person from being shown with different dates of birth on different records.

- In order to express the same facts without violating 3NF, it is necessary to split the table into two:

Tournament Winners

Tournament	**Year**	**Winner**
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Player Dates of Birth

Player	**Date of Birth**
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

- Update anomalies cannot occur in these tables, which are both in 3NF.

8.4 FOURTH NORMAL FORM

Fourth normal form (**4NF**) is a normal form used in database normalization. Introduced by Ronald Fagin in 1977, 4NF is the next level of normalization after Boyce-Codd normal form (BCNF). Whereas the second, third, and Boyce-Codd normal forms are concerned with functional dependencies, 4NF is concerned with a more general type of dependency known as a multivalued dependency. A table is in 4NF if and only if, for every one of its non-trivial multivalued dependencies $X \rightarrow\rightarrow Y$, X is a superkey—that is, X is either a candidate key or a superset thereof.

Consider the following example:

Pizza Delivery Permutations

Restaurant	Pizza Variety	Delivery Area
A1 Pizza	Thick Crust	Springfield
A1 Pizza	Thick Crust	Shelbyville
A1 Pizza	Thick Crust	Capital City
A1 Pizza	Stuffed Crust	Springfield
A1 Pizza	Stuffed Crust	Shelbyville
A1 Pizza	Stuffed Crust	Capital City
Elite Pizza	Thin Crust	Capital City
Elite Pizza	Stuffed Crust	Capital City
Vincenzo's Pizza	Thick Crust	Springfield
Vincenzo's Pizza	Thick Crust	Shelbyville
Vincenzo's Pizza	Thin Crust	Springfield
Vincenzo's Pizza	Thin Crust	Shelbyville

Each row indicates that a given restaurant can deliver a given variety of pizza to a given area.

The table has no non-key attributes because its only key is {Restaurant, Pizza Variety, Delivery Area}. Therefore it meets all normal forms up to BCNF. If we assume, however, that pizza varieties offered by a restaurant are not affected by delivery area, then it does not meet 4NF. The problem is that the table features two non-trivial multivalued dependencies on the {Restaurant} attribute (which is not a superkey). The dependencies are:

- {Restaurant} →→ {Pizza Variety}
- {Restaurant} →→ {Delivery Area}

These non-trivial multivalued dependencies on a non-superkey reflect the fact that the varieties of pizza a restaurant offers are independent from the areas to which the restaurant delivers. This state of affairs leads to redundancy in the table: for example, we are told three times that A1 Pizza offers Stuffed Crust, and if A1 Pizza starts producing Cheese Crust pizzas then we will need to add multiple rows, one for each of A1 Pizza's delivery areas. There is, moreover, nothing to prevent us from doing this incorrectly: we might add Cheese Crust rows for all but one of A1 Pizza's delivery areas, thereby failing to respect the multivalued dependency {Restaurant} →→ {Pizza Variety}.

To eliminate the possibility of these anomalies, we must place the facts about varieties offered into a different table from the facts about delivery areas, yielding two tables that are both in 4NF:

Varieties By Restaurant

Restaurant	Pizza Variety
A1 Pizza	Thick Crust
A1 Pizza	Stuffed Crust
Elite Pizza	Thin Crust
Elite Pizza	Stuffed Crust
Vincenzo's Pizza	Thick Crust
Vincenzo's Pizza	Thin Crust

Delivery Areas By Restaurant

Restaurant	Delivery Area
A1 Pizza	Springfield
A1 Pizza	Shelbyville
A1 Pizza	Capital City
Elite Pizza	Capital City
Vincenzo's Pizza	Springfield
Vincenzo's Pizza	Shelbyville

In contrast, if the pizza varieties offered by a restaurant sometimes did legitimately vary from one delivery area to another, the original three-column table would satisfy 4NF.

8.5 FIFTH NORMAL FORM

Fifth normal form (**5NF**), also known as **Project-join normal form** (**PJ/NF**) is a level of database normalization, designed to reduce redundancy in relational databases recording multi-valued facts by isolating semantically related multiple relationships. A table is said to be in the 5NF if and only if every join dependency in it is implied by the candidate keys.

A join dependency *{A, B, … Z} on R is implied by the candidate key(s) of R if and only if each of A, B, …, Z is a superkey for R
Consider the following example:

Travelling Salesman Product Availability By Brand		
Travelling Salesman	**Brand**	**Product Type**
Jack Schneider	Acme	Vacuum Cleaner
Jack Schneider	Acme	Breadbox
Willy Loman	Robusto	Pruning Shears
Willy Loman	Robusto	Vacuum Cleaner
Willy Loman	Robusto	Breadbox
Willy Loman	Robusto	Umbrella Stand
Louis Ferguson	Robusto	Vacuum Cleaner
Louis Ferguson	Robusto	Telescope
Louis Ferguson	Acme	Vacuum Cleaner
Louis Ferguson	Acme	Lava Lamp
Louis Ferguson	Nimbus	Tie Rack

The table's predicate is: Products of the type designated by *Product Type*, made by the brand designated by *Brand*, are available from the travelling salesman designated by *Travelling Salesman*.

In the absence of any rules restricting the valid possible combinations of Travelling Salesman, Brand, and Product Type, the three-attribute table above is necessary in order to model the situation correctly.

Suppose, however, that the following rule applies: *A Travelling Salesman has certain Brands and certain Product Types in his repertoire. If Brand B is in his repertoire, and Product Type P is in his repertoire, then (assuming Brand B makes Product Type P), the Travelling Salesman must offer products of Product Type P made by Brand B.*

In that case, it is possible to split the table into three:

Product Types By Travelling Salesman

Travelling Salesman	**Product Type**
Jack Schneider	Vacuum Cleaner
Jack Schneider	Breadbox
Willy Loman	Pruning Shears
Willy Loman	Vacuum Cleaner
Willy Loman	Breadbox
Willy Loman	Umbrella Stand
Louis Ferguson	Telescope
Louis Ferguson	Vacuum Cleaner
Louis Ferguson	Lava Lamp
Louis Ferguson	Tie Rack

Brands By Travelling Salesman

Travelling Salesman	**Brand**
Jack Schneider	Acme
Willy Loman	Robusto
Louis Ferguson	Robusto
Louis Ferguson	Acme
Louis Ferguson	Nimbus

Product Types By Brand

Brand	**Product Type**
Acme	Vacuum Cleaner
Acme	Breadbox
Acme	Lava Lamp
Robusto	Pruning Shears
Robusto	Vacuum Cleaner
Robusto	Breadbox
Robusto	Umbrella Stand
Robusto	Telescope
Nimbus	Tie Rack

Note how this setup helps to remove redundancy.

8.6 BCNF

Boyce-Codd normal form (or **BCNF** or **3.5NF**) is a normal form used in database normalization. It is a slightly stronger version of the third normal form (3NF). A table is in Boyce-Codd normal form if and only if for every one of its non-trivial [dependencies] $X \rightarrow Y$, X is a superkey—that is, X is either a candidate key or a superset thereof.

Only in rare cases does a 3NF table not meet the requirements of BCNF. A 3NF table which does not have multiple overlapping candidate keys is guaranteed to be in BCNF.[4] Depending on what its functional dependencies are, a 3NF table with two or more overlapping candidate keys may or may not be in BCNF.

An example of a 3NF table that does not meet BCNF is:

Today's Court Bookings

Court	Start Time	End Time	Rate Type
1	09:30	10:30	SAVER
1	11:00	12:00	SAVER
1	14:00	15:30	STANDARD
2	10:00	11:30	PREMIUM-B
2	11:30	13:30	PREMIUM-B
2	15:00	16:30	PREMIUM-A

- Each row in the table represents a court booking at a tennis club that has one hard court (Court 1) and one grass court (Court 2)
- A booking is defined by its Court and the period for which the Court is reserved
- Additionally, each booking has a Rate Type associated with it. There are four distinct rate types:
  - SAVER, for Court 1 bookings made by members
  - STANDARD, for Court 1 bookings made by non-members
  - PREMIUM-A, for Court 2 bookings made by members
  - PREMIUM-B, for Court 2 bookings made by non-members

The table's candidate keys are:

- {Court, Start Time}
- {Court, End Time}
- {Rate Type, Start Time}
- {Rate Type, End Time}

Recall that 2NF prohibits partial functional dependencies of non-prime attributes on candidate keys, and that 3NF prohibits transitive functional dependencies of non-prime attributes on candidate keys. In the Today's Court Bookings table, there are no non-prime attributes: that is, all attributes belong to candidate keys. Therefore the table adheres to both 2NF and 3NF.

The table does not adhere to BCNF. This is because of the dependency Rate Type → Court, in which the determining attribute (Rate Type) is neither a candidate key nor a superset of a candidate key.

Any table that falls short of BCNF will be vulnerable to logical inconsistencies. In this example, enforcing the candidate keys will not ensure that the dependency Rate Type → Court is respected. There is, for instance, nothing to stop us from assigning a PREMIUM A Rate Type to a Court 1 booking as well as a Court 2 booking—a clear contradiction, as a Rate Type should only ever apply to a single Court.

The design can be amended so that it meets BCNF:

Rate Types

Rate Type	Court	Member Flag
SAVER	1	Yes
STANDARD	1	No
PREMIUM-A	2	Yes
PREMIUM-B	2	No

Today's Bookings

Court	Start Time	End Time	Member Flag
1	09:30	10:30	Yes
1	11:00	12:00	Yes
1	14:00	15:30	No
2	10:00	11:30	No
2	11:30	13:30	No
2	15:00	16:30	Yes

The candidate keys for the Rate Types table are {Rate Type} and {Court, Member Flag}; the candidate keys for the Today's Bookings table are {Court, Start Time} and {Court, End Time}. Both tables are in BCNF. Having one Rate Type associated with two different Courts is now impossible, so the anomaly affecting the original table has been eliminated.

8.7 COMPARISON OF 3NF AND BCNF

Boyce Codd normal form (also known as BCNF) is a normal form –that is a form that provides criteria for determining a table’s degree of vulnerability to logical inconsistencies and anomalies. This normal form is used in database normalisation. It is a bit stronger than its predecessor, the third normal form (also known as 3NF). A table is thought to be in BCNF if and only if for every one if

its non-trivial functional dependencies –that is a boundary that is set between two sets of attributes in a relation taken from a database– is a superkey (a set of attributes of a relational variable that postulates that in all relations assigned to that specific variable there are no two distinct rows containing the same value for the attributes in that particular set). BCNF postulates that any table that fails to meet the criteria to be attributed as a BNCF is vulnerable to logical inconsistencies.

3NF is a normal form that is also used in database normalisation. It is thought that a table is in 3NF if and only if 1) the table is in second normal form (or 2NF, which is a first normal code, or 1NF, that has met the criteria to become a 2NF), and 2) every non-prime attribute of the table is non-transitively dependent on every key of the table (meaning it is not directly dependent on every key). There is another postulation of 3NF that is also used to define the differences between 3NF and the BCNF.

This theorem was conceived by Carlo Zaniolo in 1982. It states that a table is in 3NF if and only if for each functional dependency where X → A, at least one of three conditions must hold: either X → A, X is a superkey, or A is a prime attribute (which means A is contained within a candidate key –or a minimal superkey for that relation). This newer definition differs from the theorem of a BCNF in that the latter model would simply eliminate the last condition. Even as it acts as a newer version of the 3NF theorem, there is a derivation of the Zaniolo theorem. It states that X → A is non-trivial. If that is true, let A be a noon-key attribute and also let Y be a key of R. If that holds then Y → X. This means that A is not transitively dependent on Y if and only if X → Y (or if X is a superkey.

Summary:

BCNF is a normal form in which for every one of a table's non-trivial functional dependencies, is a superkey; 3NF is normal form in which the table is in 2NF and every non-prime attribute is non-transitively dependent on every key in the table.

8.8 LOSSLESS AND DEPENDENCY PRESERVING DECOMPOSITION

8.8.1 Lossless-Join Decomposition

Let R be a relation schema and let F be H, set of FDs over R. A decomposition of R into two schemas with attribute sets X and Y is said to be a lossless-join decomposition with respect to F if, for every instance T of R that satisfies the dependencies in $F, \pi_X(r) \bowtie \pi_Y(r) = T$. In other words, we can recover the original relation from the decomposed relations.

This definition can easily be extended to cover a decomposition of Ii into In more than two relations. It is easy to see that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$ always holds.

In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, We typically obtain the tuples that were not in the original relation.

8.9 CLOSURE OF DEPENDENCIES

1. We need to consider all functional dependencies that hold. Given a set F of functional dependencies, we can prove that certain other ones also hold. We say these ones are **logically implied** by F.

2. Suppose we are given a relation scheme R = (A, B, C, G, H, I), and the set of functional dependencies:
A → B
A → C
CG → H
CG → I
B → H

Then the functional dependency A → H is logically implied.

3. To see why, let t1 and t2 be tuples such that t1[A] = t2[A]
As we are given A → B , it follows that we must also have t1[B] = t2[B]
Further, since we also have B → H , we must also have t1[H] = t2[H]
Thus, whenever two tuples have the same value on A, they must also have the same value on H, and we can say that A → H .

4. The **closure** of a set F of functional dependencies is the set of all functional dependencies logically implied by F.

5. We denote the closure of F by F+.

6. To compute F+, we can use some rules of inference called

Armstrong's Axioms:
Reexivity rule: if α is a set of attributes and β ∈α , then α→β holds.

- **Augmentation rule**: if α→β holds, and γ is a set of attributes, then γα →γβ holds.
- **Transitivity rule**: if α→β holds, and β→γ holds, then α→γ holds.

7. These rules are **sound** because they do not generate any incorrect functional dependencies. They are also **complete** as they generate all of F+.

8. To make life easier we can use some additional rules, derivable from Armstrong's Axioms:

- **Union rule**: if α→β and α→γ , then α→βγ holds.
- **Decomposition rule**: if α→β holds, then α→β and α→γ both hold.
- **Pseudotransitivity rule**: if α→β holds, and γβ→δ holds, then αγ→δ holds.
-

9. Applying these rules to the scheme and set F mentioned above, we can derive the following:

- A → H, as we saw by the transitivity rule.
- CG → HI by the union rule.
- AG → I by several steps:
  - Note that A → C holds.
  - Then AG → CG, by the augmentation rule.
  - Now by transitivity, AG → I.

8.10 MINIMAL CLOSURE (COVER)

A minimal cover for a set F of FDs is a set G of FDs such that:

1. Every dependency in G is of the forIn $X \rightarrow A$, where A is a single attribute.
2. The closure $F+$ is equal to the closure (;+.
3. If we obtain a set II of dependencies frorn G by deleting one or 1110re dependencies or by deleting attributes frorn a dependency in G, then $p+' \neq II+$.

Intuitively, a minimal cover for a set }-' of FDs is an equivalent set of dependencies that is minimal in two respects: (1) Every dependency is as small as possible; that its each attribute on the left side is necessary and the right side is a single attribute. (2) Every dependency in it is required for the closure to be equal to F^{+}.
As an example, let F be the set of dependencies:

$$\text{it} \rightarrow B,\ ABCD \rightarrow E,\ EF \rightarrow G,\ F \rightarrow I, \text{and } CDF \rightarrow EG.$$

First let us rewrite it: $CDF \rightarrow BG$ so that every right side is a single attribute:

$$ACDF \rightarrow E \text{ and } ACDF \rightarrow G,$$

Next consider ACDF -> G, This dependency is implied by the following FDs:

$A \rightarrow B$, $ABCD \rightarrow E$, and $EF \rightarrow G$,

Therefore, we can delete it, Similarly, we can delete $ACDF \rightarrow E$. Next consider $ABCD \rightarrow E$, Since $A \rightarrow B$ holds, we can replace it with $ACD \rightarrow E$, (At this point, the reader should verify that each remaining FD is minimal and required,) Thus, a minimal cover for F is the set:

$A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$, and $EF \rightarrow H$,

Summary of Schema Refinement

- If a relation is in BCNF, it is free of redundancies that can be detected using FDs. Thus, trying to ensure that all relations are in BCNF is a good heuristic.
- If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
  - Must consider whether all FDs are preserved. If a lossless-join, dependency preserving decomposition into BCNF is not possible (or unsuitable, given typical queries), should consider decomposition into 3NF.
  - Decompositions should be carried out and/or re-examined while keeping *performance requirements* in mind.

9

TRANSACTION PROCESSING

Topics covered

9.1 TRANSACTION CONCURRENCY CONTROL

In computer science, especially in the fields of computer programming (see also concurrent programming, parallel programming), operating systems (see also parallel computing), multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible. Computer systems, both software and hardware, consist of modules, or components. Each component is designed to meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, and design methodologies to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints with related performance overhead. Operation consistency and correctness should be achieved together with operation efficiency.

Concurrency control in database management systems (DBMS), other transactional objects (objects with states accessed and modified by database transactions), and related distributed applications (e.g., Grid computing and Cloud computing) ensures that database transactions are performed concurrently without the

concurrency violating the data integrity of the respective databases. Thus concurrency control is an essential component for correctness in any system where two database transactions or more can access the same data concurrently, e.g., virtually in any general-purpose database system. A database transaction is defined as an object that meets the ACID rules described below when executed. A DBMS usually guarantees that only *serializable* transaction schedules (i.e., schedules that are equivalent to *serial* schedules, where transactions are executed serially, one after another, with no overlap in time; have the *serializability* property) are generated, for correctness (unless Serializability is intentionally relaxed). For maintaining correctness in cases of failed transactions (which can always happen) schedules also need to be *recoverable* (have the *recoverability* property). A DBMS also guarantees that no effect of *committed* transactions is lost, and no effect of *aborted* (rolled back) transactions remains in the related database (maintains transactions' *atomicity*, i.e., "all or nothing" semantics).

9.2 RECOVERY OF TRANSACTION FAILURE

Process of restoring database to a correct state in the event of a failure.

Need for Recovery Control

• Two types of storage: volatile (main memory) and nonvolatile.

• Volatile storage does not survive system crashes.

• Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.

9.2.1 Transaction and recovery:

• Transactions represent basic unit of recovery.

• Recovery manager responsible for atomicity and durability.

• If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to redo (roll forward) transaction's updates.

• If transaction had not committed at failure time, recovery manager has to undo (rollback) any effects of that transaction for atomicity.

• Partial undo - only one transaction has to be undone.

• Global undo - all transactions have to be undone.

9.2.1.1 DBMS should provide following facilities to assist with recovery:

• Backup mechanism, which makes periodic backup copies of database.

• Logging facilities, which keep track of current state of transactions and database changes.

• Checkpoint facility, which enables updates to database in progress to be made permanent.

• Recovery manager, which allows DBMS to restore the database to a consistent state following a failure.

9.2.2 Three main recovery techniques:

If database has been damaged we need to restore last backup copy of database and reapply updates of committed transactions using log file.

If database is only inconsistent we need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.

The Recovery techniques are as follows:

⇨ **Deferred Update**

• Updates are not written to the database until after a transaction has reached its commit point.

• If transaction fails before commit, it will not have modified database and so no undoing of changes required.

• May be necessary to redo updates of committed transactions as their effect may not have reached database.

⇨ **Immediate Update**

• Updates are applied to database as they occur.

• Need to redo updates of committed transactions following a failure.

• May need to undo effects of transactions that had not committed at time of failure.

• Essential that log records are written before write to database. Write-ahead log protocol.

• If no "transaction commit" record in log, then that transaction was active at failure and must be undone.

• Undo operations are performed in reverse order in which they were written to log.

⇨ **Shadow Paging.**

• Maintain two page tables during life of a transaction: current page and shadow page table.

• When transaction starts, two pages are the same.

• Shadow page table is never changed thereafter and is used to restore database in event of failure.

• During transaction, current page table records all updates to database.

• When transaction completes, current page table becomes shadow page table.

9.3 SERIALIZABILITY

In concurrency control of *databases*, *transaction processing* (*transaction management*), and various transactional applications, both centralized and distributed, a transaction schedule is serializable, has the Serializability property, if its outcome (the resulting database state, the values of the database's data) is equal to the outcome of its transactions executed serially, i.e., sequentially without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems.

The rationale behind serializability is the following:

If each transaction is correct by itself, i.e., meets certain integrity conditions, then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions): "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e., complete *isolation* between each other exists. Any order of the transactions is legitimate, if no dependencies among them exist, which is assumed (see comment below). As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Schedules that are not serializable are likely to generate erroneous outcomes. Well known examples are with transactions that debit and credit accounts with money: If the related schedules are not serializable, then the total sum of money may not be preserved. Money could disappear, or be generated from nowhere. This and violations of possibly needed other invariant preservations are caused by one transaction writing, and "stepping on" and erasing what has been written by another transaction before it has become permanent in the database. It does not happen if serializability is maintained.

9.4 LOG BASED RECOVERY

The commit or rollback that the DBMS performs is with the help of a file which keeps track of old and new records known as transaction log. All the DBMS brands make use of transaction log. When a user executes a SQL statement that modifies the database, the DBMS automatically writes a record in the transaction log showing two copies of each row affected by the statement; 1) one copy shows the row before the change and the other copy shows the row after the change.2) only after the log file is written does the DBMS automatically modifies the row on the disk.3)If commit occurs the new record is chosen by the log file or else if rollback occurs, the old record is chosen.

If a system failure occurs, the system operator typically recovers the data by running a special recovery utility supplied with the DBMS. The recovery utility examines the end of the transaction log, looking for the transactions that were not committed before failure. The utility rolls back each of these incomplete transactions so that only committed transactions are reflected in the data base.

9.5 LOCKING TECHNIQUES

Virtually all major DBMS products use sophisticated locking techniques to handle concurrent SQL transactions for many simultaneous users.

When transaction A accesses the database, the DBMS automatically locks each piece of the database that the transaction retrieves or modifies. Transaction B proceeds in parallel, and the DBMS also locks the pieces of the database that it accesses. If Transaction B tries to access part of the database that has been locked by Transaction A, the DBMS blocks Transaction B, causing it to wait for the data to be unlocked. The DBMS releases the locks held by Transaction A only when it ends in a COMMIT or ROLLBACK operation. The DBMS then "unblocks" Transaction B, allowing it to proceed. Transaction B can now lock that piece of the database on its own behalf, protecting it from the effects of other transactions.

9.5.1 Shared and Exclusive Locks

To increase concurrent access to a database, most commercial DBMS products use a locking scheme with more than one type of lock. A scheme using shared and exclusive locks is quite common:

• A shared lock is used by the DBMS when a transaction wants to read data from the database. Another concurrent transaction can

also acquire a shared lock on the same data, allowing the other transaction to also read the data.

• An exclusive lock is used by the DBMS when a transaction wants to update data in the database. When a transaction has an exclusive lock on some data, other transactions cannot acquire any type of lock (shared or exclusive) on the data.

The locking technique temporarily gives a transaction exclusive access to a piece of a database, preventing other transactions from modifying the locked data. Locking thus solves all of the concurrent transaction problems. It prevents lost updates, uncommitted data, and inconsistent data from corrupting the database. However, locking introduces a new problem—it may cause a transaction to wait for a long time while the pieces of the database that it wants to access are locked by other transactions.

9.5.2 Locking parameters

Typical parameters are as follows:

• **Lock size**. Some DBMS products offer a choice of table-level, page-level, row-level, and other lock sizes. Depending on the specific application, a different size lock may be appropriate.

• **Number of locks**. A DBMS typically allows each transaction to have some finite number of locks. The database administrator can often set this limit, raising it to permit more complex transactions or lowering it to encourage earlier lock escalation.

• **Lock escalation**. A DBMS will often automatically "escalate" locks, replacing many small locks with a single larger lock (for example, replacing many page-level locks with a table-level lock). The database administrator may have some control over this escalation process.

• **Lock timeout**. Even when a transaction is not deadlocked with another transaction, it may wait a very long time for the other transaction to release its locks. Some DBMS brands implement a timeout feature, where a SQL statement fails with a SQL error code if it cannot obtain the locks it needs within a certain period of time. The timeout period can usually be set by the database administrator.

9.6 GRANULARITY IN LOCKS

An important property of a lock is its **granularity**. The granularity is a measure of the amount of data the lock is protecting. In general, choosing a coarse granularity (a small

number of locks, each protecting a large segment of data) results in less **lock overhead** when a single process is accessing the protected data, but worse performance when multiple processes are running concurrently. This is because of increased **lock contention**. The more coarse the lock, the higher the likelihood that the lock will stop an unrelated process from proceeding. Conversely, using a fine granularity (a larger number of locks, each protecting a fairly small amount of data) increases the overhead of the locks themselves but reduces lock contention. More locks also increase the risk of deadlock.

9.6.1 Levels of locking

⇨ **Database level locking:**

Locking can be implemented at various levels of the database. In its crudest form, the DBMS could lock the entire database for each transaction. This locking strategy would be simple to implement, but it would allow processing of only one transaction at a time. Table level locking:

In this scheme, the DBMS locks only the tables accessed by a transaction. Other transactions can concurrently access other tables. This technique permits more parallel processing, but still leads to unacceptably slow performance in applications such as order entry, where many users must share access to the same table or tables.

⇨ **Page level locking:**

Many DBMS products implement locking at the page level. In this scheme, the DBMS locks individual blocks of data ("pages") from the disk as they are accessed by a transaction. Other transactions are prevented from accessing the locked pages but may access (and lock for themselves) other pages of data. Page sizes of 2KB, 4KB, and 16KB are commonly used. Since a large table will be spread out over hundreds or thousands of pages, two transactions trying to access two different rows of a table will usually be accessing two different pages, allowing the two transactions to proceed in parallel.

⇨ **Row level locking:**

Over the last several years, most of the major commercial DBMS systems have moved beyond page-level locking to row-level locks. Row-level locking allows two concurrent transactions that access two different rows of a table to proceed in parallel, even if the two rows fall in the same disk block. While this may seem a remote possibility, it can be a real problem with small tables containing small records. Row-level locking provides a high degree of parallel transaction execution. Unfortunately, keeping track of locks

on variable-length pieces of the database (in other words, rows) rather than fixed-size pages is a much more complex task, so increased parallelism comes at the cost of more sophisticated locking logic and increased overhead.

9.7 TIME STAMPING TECHNIQUES

A trusted timestamp is a timestamp issued by a trusted third party (TTP) acting as a **time stamping authority** (**TSA**). It is used to prove the existence of certain data before a certain point (e.g. contracts, research data, medical records,...) without the possibility that the owner can backdate the timestamps. Multiple TSAs can be used to increase reliability and reduce vulnerability.

9.7.1 Creating a timestamp

The technique is based on digital signatures and hash functions. First a hash is calculated from the data. A hash is a sort of digital fingerprint of the original data: a string of bits that is different for each set of data. If the original data is changed then this will result in a completely different hash. This hash is sent to the TSA. The TSA concatenates a timestamp to the hash and calculates the hash of this concatenation. This hash is in turn digitally signed with the private key of the TSA. This signed hash + the timestamp is sent back to the requester of the timestamp who stores these with the original data (see diagram).

Since the original data can not be calculated from the hash (because the hash function is a one way function), the TSA never gets to see the original data, which allows the use of this method for confidential data.

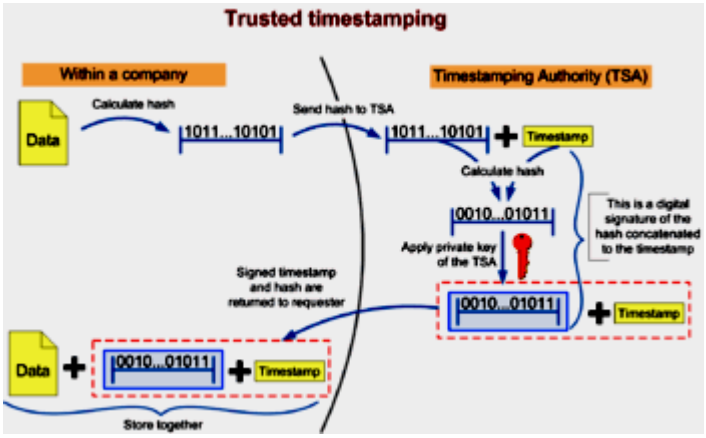


Figure: Getting a timestamp from a trusted third party.

9.7.2 Checking the timestamp

Anyone trusting the timestamper can then verify that the document was *not* created *after* the date that the timestamper vouches. It can also no longer be repudiated that the requester of the timestamp was in possession of the original data at the time

given by the timestamp. To prove this (see diagram) the hash of the original data is calculated, the timestamp given by the TSA is appended to it and the hash of the result of this concatenation is calculated, call this hash A.

Then the digital signature of the TSA needs to be validated. This can be done by checking that the signed hash provided by the TSA was indeed signed with their private key by digital signature verification. The hash A is compared with the hash B inside the signed TSA message to confirm they are equal, proving that the timestamp and message is unaltered and was issued by the TSA. If not, then either the timestamp was altered or the timestamp was not issued by the TSA.

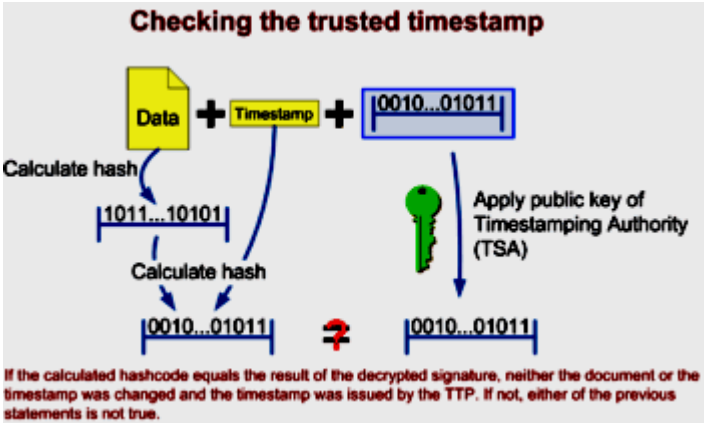


Figure: Checking correctness of a timestamp generated by a time stamping authority (TSA).

9.8 TWO PHASE LOCKING SYSTEM:

According to the **two-phase locking** protocol, a transaction handles its locks in two distinct, consecutive phases during the transaction's execution:

1. **Expanding phase** (number of locks can only increase): locks are acquired and no locks are released.
2. **Shrinking phase**: locks are released and no locks are acquired.

The serializability property is guaranteed for a schedule with transactions that obey the protocol. The 2PL *schedule class* is defined as the class of all the schedules comprising transactions with data access orders that could be generated by the 2PL protocol.

Typically, without explicit knowledge in a transaction on end of phase-1, it is safely determined only when a transaction has entered its *ready* state in all its processes (processing has ended, and it is ready to be committed; no additional locking is possible). In

this case phase-2 can end immediately (no additional processing is needed), and actually no phase-2 is needed. Also, if several processes (two or more) are involved, then a synchronization point (similar to *atomic commitment*) among them is needed to determine end of phase-1 for all of them (i.e., in the entire distributed transaction), to start releasing locks in phase-2 (otherwise it is very likely that both 2PL and Serializability are quickly violated). Such synchronization point is usually too costly (involving a distributed protocol similar to atomic commitment), and end of phase-1 is usually postponed to be merged with transaction end (atomic commitment protocol for a multi-process transaction), and again phase-2 is not needed. This turns 2PL to SS2PL (see below). All known implementations of 2PL in products are SS2PL based.

9.8.1 Strict two-phase locking

The **strict two-phase locking** (S2PL) class of schedules is the intersection of the 2PL class with the class of schedules possessing the *Strictness* property.

To comply with the S2PL protocol a transaction needs to comply with 2PL, and release its *write (exclusive)* locks only after it has ended, i.e., being either *committed* or *aborted*. On the other hand, *read (shared)* locks are released regularly during phase 2. Implementing general S2PL requires explicit support of phase-1 end, separate from transaction end, and no such widely utilized product implementation is known.

S2PL is a special case of 2PL, i.e., the S2PL class is a proper subclass of 2PL.

9.8.2 Strong strict two-phase locking or **Rigorousness**, or **Rigorous scheduling**, or **Rigorous two-phase locking**

To comply with **strong strict two-phase locking** (SS2PL) the locking protocol releases both *write (exclusive)* and *read (shared)* locks applied by a transaction only after the transaction has ended, i.e., being either *committed* or *aborted*. This protocol also complies with the S2PL rules. A transaction obeying SS2PL can be viewed as having phase-1 that lasts the transaction's entire execution duration, and no phase-2 (or a degenerate phase-2). Thus, only one phase is actually left, and "two-phase" in the name seems to be still utilized due to the historical development of the concept from 2PL, and 2PL being a super-class. The SS2PL property of a schedule is also called **Rigorousness**. It is also the name of the class of schedules having this property, and an SS2PL schedule is also called a "rigorous schedule". The term "Rigorousness" is free of the unnecessary legacy of "two-phase," as well as being independent of any (locking) mechanism (in

principle other blocking mechanisms can be utilized). The property's respective locking mechanism is sometimes referred to as **Rigorous 2PL**.

9.9 DEADLOCK HANDLING

A **deadlock** is a situation wherein two or more competing actions are each waiting for the other to finish, and thus neither ever does.

An example of a deadlock which may occur in database products is the following. Client applications using the database may require exclusive access to a table, and in order to gain exclusive access they ask for a *lock*. If one client application holds a lock on a table and attempts to obtain the lock on a second table that is already held by a second client application, this may lead to deadlock if the second application then attempts to obtain the lock that is held by the first application. (But this particular type of deadlock is easily prevented, e.g., by using an *all-or-none* resource allocation algorithm.)

There are four necessary and sufficient conditions for a Coffman deadlock to occur, known as the *Coffman conditions* from their first description in a 1971 article by E. G. Coffman.

1. Mutual exclusion condition: a resource that cannot be used by more than one process at a time
2. Hold and wait condition: processes already holding resources may request new resources
3. No preemption condition: No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process
4. Circular wait condition: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds

9.9.1 Prevention

- Removing the mutual exclusion condition means that no process may have exclusive access to a resource. This proves impossible for resources that cannot be spooled, and even with spooled resources deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.

- The "hold and wait" conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations); this advance knowledge is frequently difficult to

satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to release all their resources before requesting all the resources they will need. This too is often impractical. (Such algorithms, such as serializing tokens, are known as the all-or-none algorithms.)

- A "no preemption" (lockout) condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. (Note: Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead.) Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

- The circular wait condition: Algorithms that avoid circular waits include "disable interrupts during critical sections", and "use a hierarchy to determine a partial ordering of resources" (where no obvious hierarchy exists, even the memory address of resources has been used to determine ordering) and Dijkstra's solution.

9.9.2 Circular wait prevention

Circular wait prevention consists of allowing processes to wait for resources, but ensure that the waiting can't be circular. One approach might be to assign a precedence to each resource and force processes to request resources in order of increasing precedence. That is to say that if a process holds some resources and the highest precedence of these resources is *m*, then this process cannot request any resource with precedence smaller than *m*. This forces resource allocation to follow a particular and non-circular ordering, so circular wait cannot occur. Another approach is to allow holding only one resource per process; if a process requests another resource, it must first free the one it is currently holding (that is, disallow hold-and-wait).

9.9.3 Avoidance

Deadlock can be avoided if certain information about processes is available in advance of resource allocation. For every resource request, the system sees if granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states. In order for the system to be able to figure out whether the next state will be safe or unsafe, it must know in advance at any time the number and type of all resources in existence, available, and requested. One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which

requires resource usage limit to be known in advance. However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is often impossible.

Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetry-breaking technique. In both these algorithms there exists an older process (O) and a younger process (Y). Process age can be determined by a timestamp at process creation time. Smaller time stamps are older processes, while larger timestamps represent younger processes.

	Wait/Die	**Wound/Wait**
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

It is important to note that a process may be in an unsafe state but would not result in a deadlock. The notion of safe/unsafe states only refers to the ability of the system to enter a deadlock state or not. For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

9.9.4 Detection

Often, neither avoidance nor deadlock prevention may be used. Instead deadlock detection and process restart are used by employing an algorithm that tracks resource allocation and process states, and rolls back and restarts one or more of the processes in order to remove the deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler or OS.

Detecting the possibility of a deadlock *before* it occurs is much more difficult and is, in fact, *generally* undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in *specific* environments, using *specific* means of locking resources, deadlock detection may be *decidable*. In the *general* case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.

Deadlock detection techniques include, but are not limited to, Model checking. This approach constructs a Finite State-model on which it performs a progress analysis and finds all possible terminal sets in the model. These then each represent a deadlock.

10

SECURITY AND AUTHORIZATION

Topics covered

10.1 GRANTING PRIVILEGES: [GRANT STATEMENT]

10.2 REVOKING PREVILEGES [REVOKE STATEMENT]

10.3 PASSING PRIVILEGES: - (Using Grant option)

Implementing a security scheme and enforcing security restrictions are the responsibilities of the dbms software.

The actions that a user is allowed permitted to carry out for a given database object (i.e. forms, application programs, tables or entire database) are called privilege. Users may have permission to insert and select a row in a certain table but may lack permission to delete or update rows of that table.

To establish a security scheme on a database, you use the SQL GRANT and REVOKE statements.

10.1 GRANTING PRIVILEGES: [GRANT STATEMENT]:

The GRANT statement is used to grant the privileges on the database objects to specific users. Normally the GRANT statement is used by owner of the table or view to give other users access to the data. The GRANT statement includes list of the privileges to be granted, name of the table to which privileges apply and user id to which privileges are granted.

E.g. 1) Give user ABC full access to employee table:

GRANT Select, Insert, Delete, update on employee to ABC

2) Let user PQR only read the employee table. Update, delete and insert are not allowed.

GRANT Select on employee to PQR

3) Give all users select access to employee table:

GRANT Select on employee to public.

Note that GRANT statement in the above example grants access to all present and future authorized users. This eliminates the need for you to explicitly grant privileges to new users as they are authorized.

10.2 REVOKING PREVILEGES [REVOKE STATEMENT]:

In most SQL based databases, the privileges that you have granted with the GRANT statement can be taken away with the REVOKE statement. The structure of the REVOKE statement is much similar to that of the GRANT statement. A REVOKE statement may take away all or some of the privileges granted to a user id.

E.g. Revoke Select, Insert on employee from ABC.

10.3 PASSING PRIVILEGES: - (Using Grant option)

When you create a data base object and become it's owner, you are the only person who can grant privileges to use these objects. When you grant privileges to other users, they are allowed to use that object but can not pass those privileges on to other users. In this way, the owner of the object maintains very tight control, both over who has permission to use the object and over which form of access are allowed. Occasionally you may want to allow other users to grant privileges on an object that you own.

Ex: Consider user XYZ wants user ABC to grant privileges on emp table to any other user. So he would Grant Select on emp to ABC With Grant option

The key word 'with grant opt' enables user ABC to grant privilege on emp table to any other user, though he owns the table. so as ABC would to user PQR

'Grant Select on XYZ-emp to PQR'

10.3.1REVOKE AND GRANT OPTION:-

When you grant privileges with the GRANT OPTON and later revoke these privileges, most DBMS brands will automatically revoke all privileges derived from the original grant. Consider again the chain of privileges ABC to PQR and then PQR to XYZ. If ABC revokes PQR's privileges, then XYZ's privileges will also be revoked. The situation gets more complicated if two or more users have granted privileges and one of them later revokes the privileges.

Consider that PQR gets Grant permission from both ABC and XYZ and then grants privileges to MNO. When ABC revokes permission from PQR, then grant from XYZ will remain. Further MNO privileges will also remain because they can be derived from XYZ.