Biyani's Think Tank

Concept based notes

Database Management System

M.Sc.-IT

Bhavana Sangamnerkar

M.Sc. (Comp. Prog.), PGDCA

H.O.D.

Information Technology

Biyani Girls College, Jaipur

Published by :

Think Tanks

Biyani Group of Colleges

Sector-3, Vidhyadhar Nagar,

Jaipur-302 023 (Rajasthan)

Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007

E-mail : acad@biyanicolleges.org

Website :www.gurukpo.com; www.biyanicolleges.org

First Edition : 2009

Leaser Type Setted by :

Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the “Teach Yourself” style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, *Chairman* & Dr. Sanjay Biyani, *Director* (*Acad.*) Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this Endeavour. They played an active role in coordinating the various stages of this Endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Author

Syllabus

M.Sc.-IT (1st Sem.)

Database Management System

Data and Information [Basic Concepts, Problems of Early Information Systems, Advantages of a DBMS]

Database Architectures [Three Levels of the Architecture : External, Conceptual and Internal Level],

Centralized and Distributed

Database Models : Hierarchical [Concepts of a Hierarchy, IMS Hierarchy]

Relational [Concepts of Relational Model, Relational Algebra, Relational Calculus]

Network [Concepts of a Network, DBTG Network, DBA Schema Declaration]

Object Oriented Database [Only Basic Information about OODBMS and ORDBMS]

Database Query Languages [Basic Retrieval Capability, Retrieval and Explosion, Update Commands, QBEI, Client/ Server Design]

Standard Query Language [Basic SQL Query, Nested Queries Aggregate Operators, Null Values, Embedded SQL, Cursor, Dynamic SQL Query Optimization [Query Evaluation Plans, Pipelined Evaluation, Iterator Interface for Operators and Access Methods, Relational Query Optimizer]

Data Management Issues : Backup, Recovery, Maintenance, and Performance.

Database Design [Schema Refinement, Functional Dependencies, Normal Forms, Decompositions, Normalization]

Tuning [Tuning Indexes, Tuning Queries and Views, Tuning the Conceptual Schema, DBMS Benchmarking]

Security [Access Control, Discretionary and Mandatory Access Control, Encryption] and **Implementation**.
Enterprise Wide Data Application [Information only]

Building Client/Server Databases [Information only]

Object Oriented Databases [Information only]

Internet Databases [Information only]

Open Database Connectivity (ODBC) [Information only]

Accessing Remote Data Sources

Transaction Management [Information only].

Databases and Tools : MS-Access SQL Visual Basic ORACLE [Wherever required these tools should be used].

□ □ □

Content

□ □ □

Chapter-1

Data and Information

Q.1 What do you mean by Data and Information?

Ans.: Data are plain facts. The word "data" is plural for "datum." When data are processed, organized, structured or presented in a given context so as to make them useful, they are called Information. It is not enough to have data (such as statistics on the economy). Data themselves are fairly useless, but when these data are interpreted and processed to determine its true meaning, they becomes useful and can be named as Information.

Q.2 What do you mean by Database?

Ans.: Definitions of **Database** :

- An organized body of related information.
- In computing, a database can be defined as a structured collection of records or data that is stored in a computer so that a program can consult it to answer queries. The records retrieved in answer to queries become information that can be used to make decisions.
- An organized collection of records presented in a standardized format searched by computers.
- A collection of data organized for rapid search and retrieval by a computer.
- A collection of related data stored in one or more computerized files in a manner that can be accessed by users or computer programs via a database management system.
- An organized collection of information, data, or citations stored in electronic format that can be searched for specific information or records by techniques specific to each database.
- A logical collection of interrelated information, managed and stored as a unit, usually on some form of mass-storage system such as magnetic tape or disk.
- A database is a structured format for organizing and maintaining information that can be easily retrieved. A simple example of a database is a table or a spreadsheet.
- A database in an organized collection of computer records. The most common type of database consists of records describing articles in periodicals otherwise known as a periodical index.
- A database collects information into an electronic file, for example a list of customer addresses and associated orders. Each item is usually called a 'record' and the items can be sorted and accessed in many different ways.
- A set of related files that is created and managed by a Database Management System (DBMS).
- A computerized collection of information.
- Integrated data files organized and stored electronically in a uniform file structure that allows data elements to be manipulated, correlated, or extracted to satisfy diverse analytical and reporting needs.
- A collection of information stored in one central location. Many times, this is the source from which information is pulled to display products or information dynamically on a website.
- Relational data structure used to store, query, and retrieve information.
- An organized set of data or collection of files that can be used for a specified purpose. A collection of interrelated data stored so that it may be accessed with user friendly dialogs.
- A large amount of information stored in a computer system.

Q.3 What are the basic objectives of the Database?

Ans.: A database is a collection of interrelated data stored with minimum redundancy to serve many users quickly and efficiently. The general objective is to make information access easy, quick, inexpensive, and flexible for the user. In data base design, several **specific objectives** can be considered as follows:

(i) Controlled Redundancy

(ii) Ease of Learning and Use

(iii) Data Independence

(iv) Most Information in Low Cost

(v) Accuracy and Integrity

(vi) Recovery from failure

(vii) Privacy and Security

(viii) Performance

Q.4 Define the Database Management System.

Ans.: (i) A **Database Management System** (DBMS), or simply a **Database System** (DBS), consists of :

- A collection of interrelated and persistent data (usually referred to as the **Database** (DB)).
- A set of application programs used to access, update and manage that data (which form the data Management System (MS)).

(ii) The goal of a DBMS is to provide an environment that is both **convenient** and **efficient** to use in :

- Retrieving information from the database.
- Storing information into the database.

(iii) Databases are usually designed to manage **large** bodies of information. This involves :

- Definition of structures for information storage (data modeling).
- Provision of mechanisms for the manipulation of information (file and systems structure, query processing).
- Providing for the safety of information in the database (crash recovery and security).
- Concurrency control if the system is shared by users.

Q.5 Describe the basic components of DBMS. Why do we need DBMS.

Or

What are the Advantages of DBMS over conventional file system.

Ans.: There are four basic components of Database Management System :

(i) **Data :** Raw facts which we want to feed in the computer.

(ii) **Hardware :** On which the data to be processed.

(iii) **Software :** The interface between the hardware and user, by which the data will change into the information.

(iv) **User :** There are so many types of users some of them are application programmer, endcase users and DBA.

Purpose of Database Systems :

(i) To see why database management systems are necessary, let's look at a typical ``File-Processing System'' supported by a conventional operating system.

The application is a savings bank :

- Savings account and customer records are kept in permanent system files.
- Application programs are written to manipulate files to perform the following **tasks** :
  - Debit or credit an account.
  - Add a new account.
  - Find an account balance.
  - Generate monthly statements.

(ii) Development of the System proceeds as follows :

- New application programs must be written as the need arises.
- New permanent files are created as required.
- but over a long period of time files may be in different formats, and
- Application programs may be in different languages.

(iii) So we can see there are problems with the Straight File-Processing Approach :

- **Data Redundancy and Inconsistency :**
  - Same information may be duplicated in several places.
  - All copies may not be updated properly.
- **Difficulty in Accessing Data :**
  - May have to write a new application program to satisfy an unusual request.
  - E.g. find all customers with the same postal code.
  - Could generate this data manually, but a long job.
- **Data Isolation :**
  - Data in different files.
  - Data in different formats.
  - Difficult to write new application programs.
- **Multiple Users :**
  - Want concurrency for faster response time.
  - Need protection for concurrent updates.
  - E.g. two customers withdrawing funds from the same account at the same time - account has $500 in it, and they withdraw $100 and $50. The result could be $350, $400 or $450 if no protection.
- **Security Problems :**
  - Every user of the system should be able to access only the data they are permitted to see.
  - E.g. payroll people only handle employee records, and cannot see customer accounts; tellers only access account data and cannot see payroll data.
  - Difficult to enforce this with application programs.
- **Integrity Problems :**
  - Data may be required to satisfy constraints.
  - E.g. no account balance below $25.00.
  - Again, difficult to enforce or to change constraints with the file-processing approach.

Above all problems lead to the development of **Database Management Systems**.

Advantages :

- An organized and comprehensiveness of recording the result of the firms activities.
- A receiver of data to be used in meeting the information requirement of the MIS users.

- Reduced data redundancy.
- Reduced updating errors and increased consistency.
- Greater data integrity and independence from applications programs.
- Improved data access to users through use of host and query languages.
- Improved data security.
- Reduced data entry, storage, and retrieval costs.
- Facilitated development of new applications program.
- Standard can be enforced: Standardized stored data format is particularly desirable as an old data to interchange or migration (change) between the system.
- Conflicting requirement can be handled.

Disadvantages :

- It increases opportunity for person or groups outside the organization to gain access to information about the firms operation.
- It increases opportunity for fully training person within the organization to misuse the data resources intentionally.
- The data approach is a costly due to higher H/W and S/W requirements.
- Database systems are complex (due to data independence), difficult, and time-consuming to design.
- It is not maintain for all organizations .It is only efficient for particularly large organizations.
- Damage to database affects virtually all applications programs.
- Extensive conversion costs in moving form a file-based system to a database system.
- Initial training required for all programmers and users.

Q.6 Define the Overall System Structure of Database Management System.

Ans.: Overall System Structure :

a) Database systems are partitioned into modules for different functions. Some functions (e.g. file systems) may be provided by the operating system.

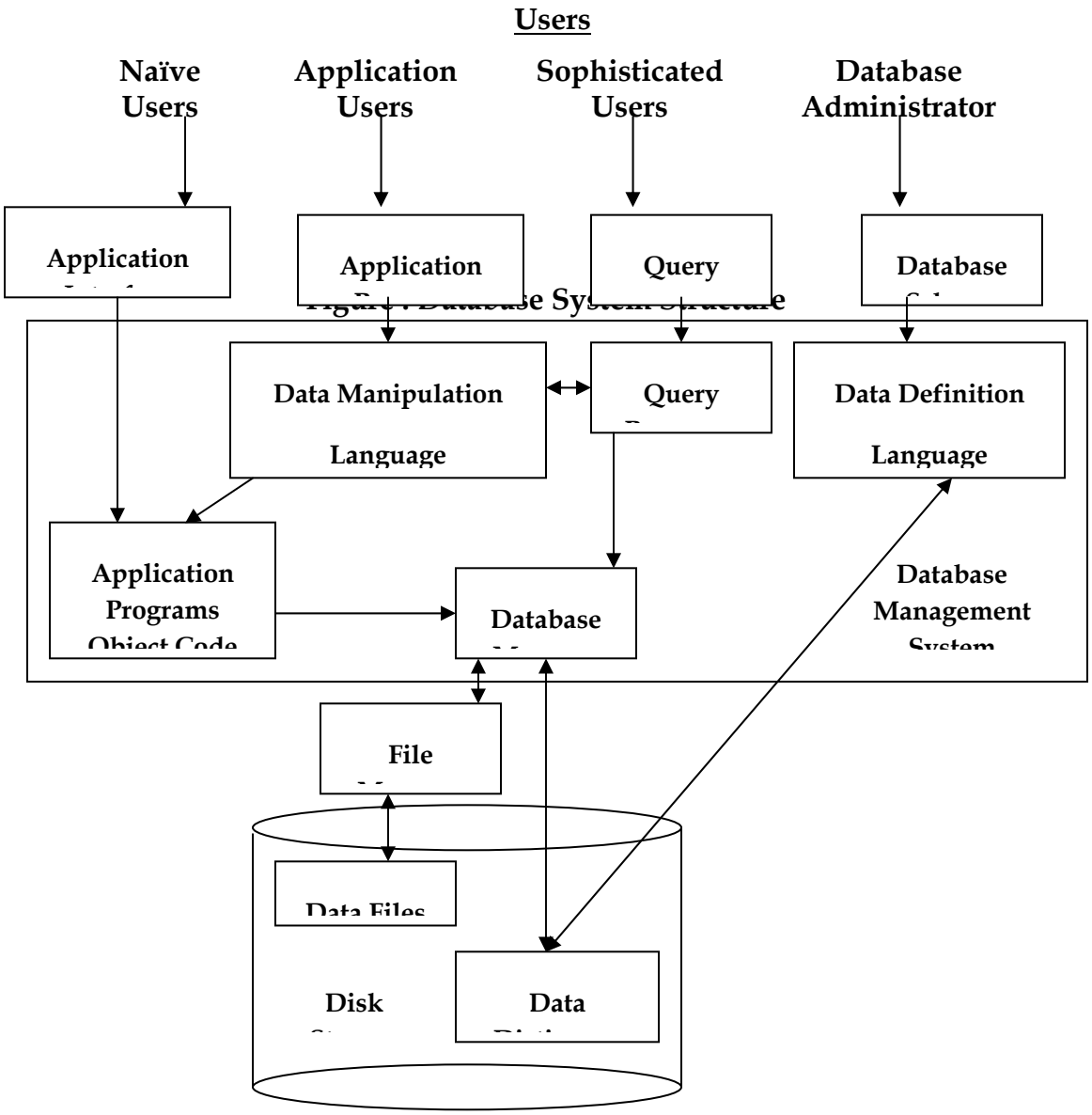
Components include :

- **File Manager** manages allocation of disk space and data structures used to represent information on disk.
- **Database Manager** : The interface between low-level data and application programs and queries.

- **Query Processor** translates statements in a query language into low-level instructions the database manager understands. (May also attempt to find an equivalent but more efficient form.)
- **DML Precompiler** converts DML statements embedded in an application program to normal procedure calls in a host language. The precompiler interacts with the query processor.
- **DDL Compiler** converts DDL statements to a set of tables containing metadata stored in a data dictionary.

In addition, several data structures are required for physical system implementation :

- **Data Files :** store the database itself.
- **Data Dictionary :** stores information about the structure of the database. It is used **heavily**. Great emphasis should be placed on developing a good design and efficient implementation of the dictionary.
- **Indices :** provide fast access to data items holding particular values.


Users
Naïve Users
Application Users
Sophisticated Users
Database Administrator
Application
Application
Query
Database
Data Manipulation Language
Query
Data Definition Language
Application Programs
Database
Database Management
File
Data Files
Disk
Data

Chapter-2

Database Architecture

Q.1 What do you mean by Data Abstraction?

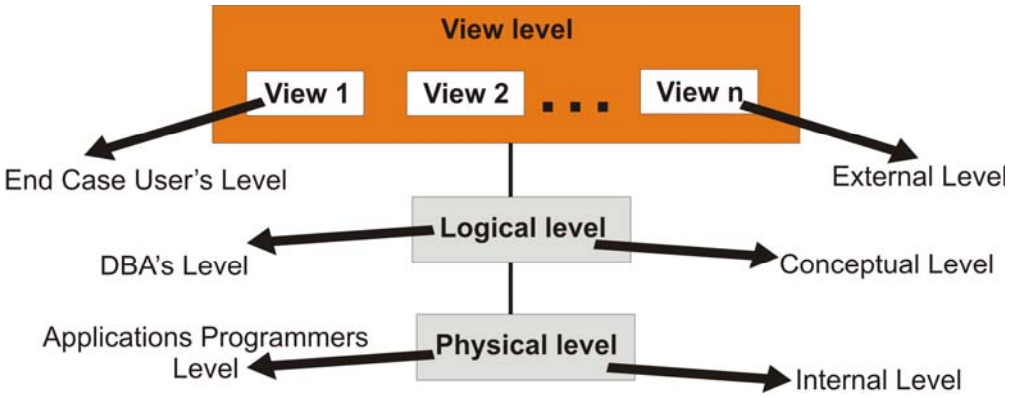
Ans.: The major purpose of a database system is to provide users with an **abstract view** of the system. The system hides certain details of how data is stored, created and maintained. Complexity should be hidden from the database users, which is known as Data Abstraction.

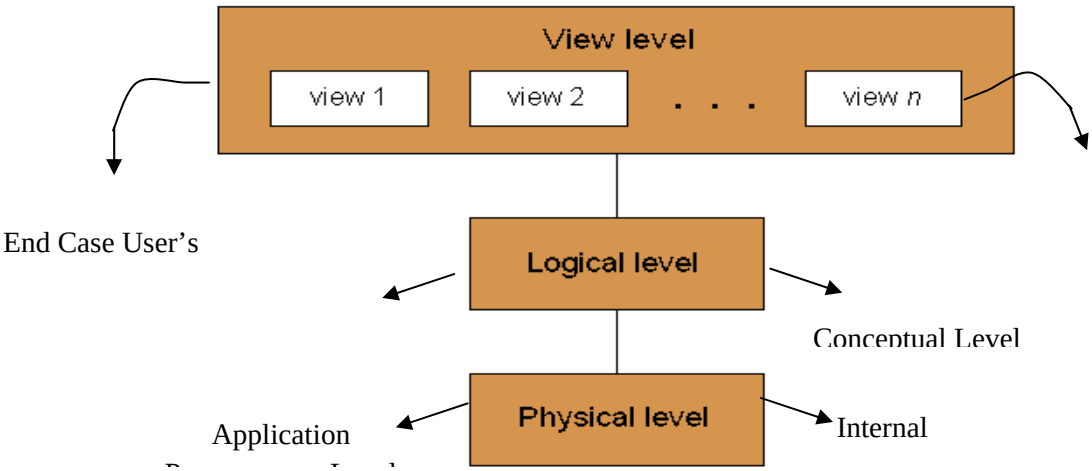
Levels of Abstraction :

A common concept in computer science is *levels* (or less commonly *layers*) of abstraction, where in each level represents a different model of the same information and processes, but uses a system of expression involving a unique set of objects and compositions that are applicable only to a particular domain. Each relatively abstract, a "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. For example, gates build on electronic circuits, binary on gates, machine language on binary, programming language on machine language, applications and operating systems on programming languages. Each level is embeded, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Q.2 Explain the Database Architecture with its various levels.

Ans.: Many users of database systems are not deeply familiar with computer data structures, database developers often hide complexity through the following levels :





Physical Level : The lowest level of abstraction describes *how* the data is actually stored. The physical level describes complex low-level data structures in detail.

Logical Level : The next higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.

View Level : The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large

database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Q.3 How will you define different Database Users in Database Management System?

Ans.: The **Database Users** fall into several categories :

- **Application Programmers** are computer professionals interacting with the system through DML calls embedded in a program written in a host language (e.g. C, PL/1, Pascal) :
  - These programs are called **Application Programs**.
  - The **DML Precompiler** converts DML calls (prefaced by a special character like $, #, etc.) to normal procedure calls in a host language.
  - The host language compiler then generates the object code.
  - Some special types of programming languages combine Pascal-like control structures with control structures for the manipulation of a database.
  - These are sometimes called **Fourth-Generation Languages**.
  - They often include features which to generate forms and display data.
- **Sophisticated Users** interact with the system without writing programs :
  - They form requests by writing queries in a database query language.
  - These are submitted to a query processor that breaks a DML statement down into instructions for the database manager module.
- **Specialized Users** are sophisticated users writing special database application programs. These may be CADD systems, knowledge-based and expert systems, complex data systems (audio/video), etc.
- **Naive Users** are unsophisticated users who interact with the system by using permanent application programs (e.g. automated teller machine).

□ □ □

Chapter-3

Data Models

Q.1 Define various Data Models.

Ans.: **Data Models:** Data models are a collection of conceptual tools for describing data, data relationships, data semantics and data constraints. There are three different groups :

(i) Object-Based Logical Models.

(ii) Record-Based Logical Models.

(iii) Physical Data Models.

We'll look at them in more detail now :

(i) Object-Based Logical Models :

- Describe data at the conceptual and view levels.
- Provide fairly flexible structuring capabilities.
- Allow one to specify data constraints explicitly.
- Over 30 such models, including :
  a) Entity-Relationship Model
  b) Object-Oriented Model
  c) Binary Model
  d) Semantic Data Model
  e) Info Logical Model
  f) Functional Data Model

*** At this point, we'll take a closer look at the **Entity-Relationship (E-R)** and **Object-Oriented** models.

a) **The E-R Model :** The entity-relationship model is based on a perception of the world as consisting of a collection of basic objects (entities) and relationships among these objects :

- An **entity** is a distinguishable object that exists.
- Each entity has associated with it a set of **attributes** describing it.
- E.g. *number* and *balance* for an account entity.

- A **relationship** is an association among several entities.
- E.g. A *cust_acct* relationship associates a customer with each account he or she has.
- The set of all entities or relationships of the same type is called the **entity set** or **relationship set**.
- Another essential element of the E-R diagram is the **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set.

The overall logical structure of a database can be expressed graphically by an **E-R diagram** :

- **Rectangles**: represent entity sets.
- **Ellipses**: represent attributes.
- **Diamonds**: represent relationships among entity sets.
- **Lines**: link attributes to entity sets and entity sets to relationships.

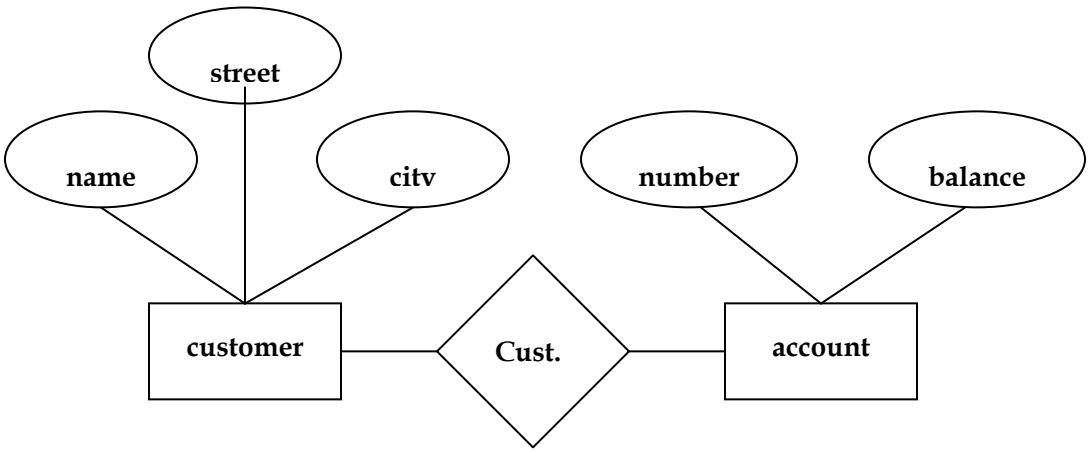


Figure : Entity-Relationship Model

b) **Object-Oriented Model :**

- The object-oriented model is based on a collection of objects, like the E-R model.
  - An object contains values stored in **instance variables** within the object.
  - Unlike the record-oriented models, these values are themselves objects.
  - Thus objects contain objects to an arbitrarily deep level of nesting.
  - An object also contains bodies of code that operate on the object.
  - These bodies of code are called **methods**.
  - Objects that contain the same types of values and the same methods are grouped into **classes**.
  - A class may be viewed as a type definition for objects.
  - Analogy: the programming language concept of an abstract data type.
  - The only way in which one object can access the data of another object is by invoking the method of that other object.
  - This is called **sending a message** to the object.
  - Internal parts of the object, the instance variables and method code, are not visible externally.
  - Result is two levels of data abstraction.
- For example, consider an object representing a bank account.
  - The object contains instance variables number and balance.
  - The object contains a method pay-interest which adds interest to the balance.
  - Under most data models, changing the interest rate entails changing code in application programs.
  - In the object-oriented model, this only entails a change within the pay-interest method.
- Unlike entities in the E-R model, each object has its own unique identity, independent of the values it contains :
  - Two objects containing the same values are distinct.
  - Distinction is created and maintained in physical level by assigning distinct object identifiers

(ii) Record-Based Logical Models :

- Also describe data at the conceptual and view levels.
- Unlike object-oriented models, they are used to
  - Specify overall logical structure of the database, and
  - Provide a higher-level description of the implementation.
- Named so because the database is structured in fixed-format records of several types.
- Each record type defines a fixed number of fields, or attributes.
- Each field is usually of a fixed length (this simplifies the implementation).
- Record-based models do not include a mechanism for direct representation of code in the database.
- Separate languages associated with the model are used to express database queries and updates.
- The three most widely-accepted models are the **relational, network**, and **hierarchical**.
- This course will concentrate on the **relational** model.
- The **network** and **hierarchical** models are covered in appendices in the text.

a) The Relational Model :

- Data and relationships are represented by a collection of **tables**.
- Each **table** has a number of columns with unique names, e.g. *customer, account*.
- Following tabler shows a sample relational database.

name	street	city	number
Lowery	Maple	Queens	900
Shiver	North	Bronx	556
Shiver	North	Bronx	647
Hodges	Sidehill	Brooklyn	801
Hodges	Sidehill	Brooklyn	647

number	balance
900	55
556	10000
647	105366
801	10533

Figure : A Sample Relational Database.

b) The Network Model :

- Data are represented by collections of records.
- Relationships among data are represented by links.
- Organization is that of an **arbitrary graph**.
- Following figure shows a sample network database that is the equivalent figure of the relational database.

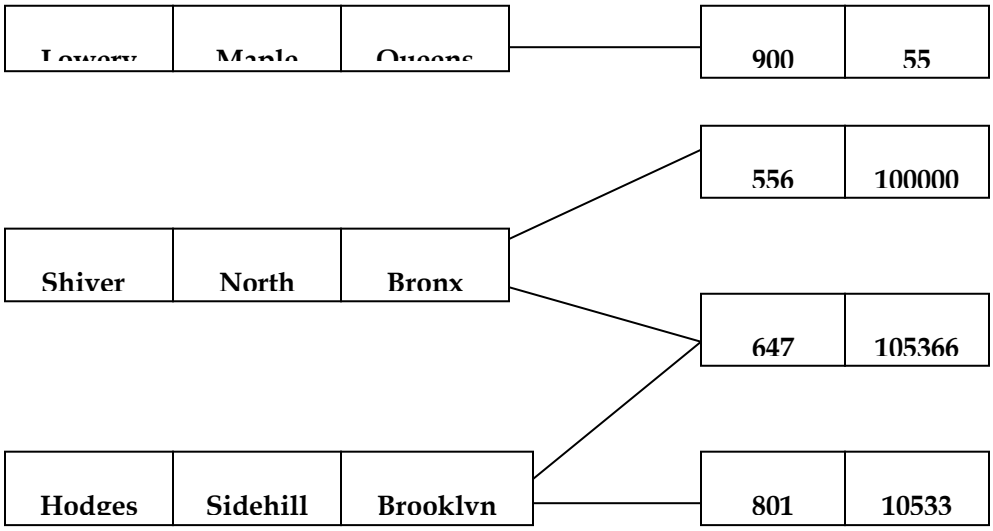


Figure : Sample Network Database

c) The Hierarchical Model :

- Similar to the network model.
- Organization of the records is as a collection of **trees**, rather than arbitrary graphs.
- Following figure shows a sample hierarchical database that is the equivalent figure of the relational database

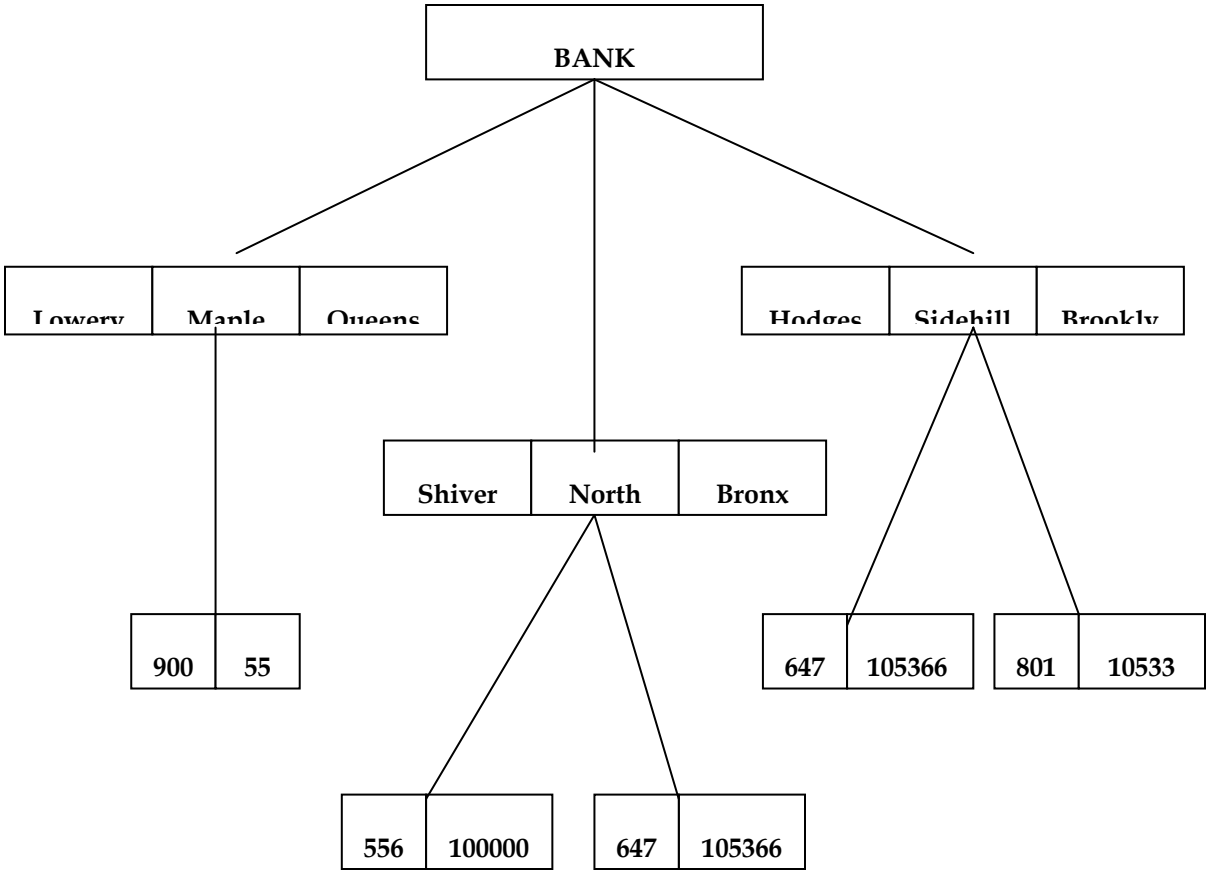


Figure : A Sample Hierarchical Database

The relational model does not use pointers or links, but relates records by the values they contain. This allows a formal mathematical foundation to be defined.

(iii) Physical Data Models :

a) Are used to describe data at the lowest level.

b) Very few models, e.g.

- Unifying model
- Frame memory

Q.2 Define the components of Data Models.

Ans.: A data model underlines the structure of a database and defines the relationship between the different entities, a data model consist of three main components.

(i) **Structures :** The structure portion specifies hoe elementary data item are grouped into layers.

(ii) **Operation :** The operation component provides mechanism for inaction relation, retrieval & modification of data

(iii) **Constraints :** The constraint define conditions that must be met for the data to be complete & correct. Constraints are used to control the condition under which a particular data object may exist be altered & do forth.

Three major types of constraints are on values depends & referential integrity :–

Constraints	**Description**
Values	The allowable, valid values for attribute may be States as a list, a range, type of character etc. For Values may be 1,2 or 3 only range from 0 to 60 or be 1,2 or 3 only range from 0 to 60 or be numeric only.
Dependencies	The allowable values for attribute may depend on it other values for the attribute may depend on it other values. For Ex. The Employee eligibility for overtime is depending upon his/her other status code.
Relational Integrity	Entity and relationship often have reference condition that must be met for e.g. there may be existence dependencies in which for one entity to exist. An illustration of this is sales order, for an order to exist there must be a customer.

Q.3 Explain IMS Hierarchy in terms of DBMS?

Ans.: **IMS Hierarchy :** IMS (Information Management System) is a database and transaction management system that was first introduced by IBM in 1968. Since then, IMS has gone through many changes in adapting to new programming tools and environments. IMS is one of two major legacy database and transaction management subsystems from IBM that run on mainframe MVS (now z/OS) systems. The other is CICS (Customer information service system). It is claimed that, historically, application programs that use either (or both) IMS or CICS services have handled and continue to handle most of the world's banking, insurance, and order entry transactions.

IMS consists of two major components, the IMS Database Management System (IMS DB) and the IMS Transaction Management System (IMS TM). In IMS DB, the data is organized into a hierarchy. The data in each level is dependent on the data in the next higher level. The data is arranged so that its integrity is ensured, and the storage and retrieval process is optimized. IMS TM controls I/O (input/output) processing, provides formatting, logging, and recovery of messages, maintains communications

security, and oversees the scheduling and execution of programs. TM uses a messaging mechanism for queuing requests. IMS's original programming interface was DL/1 (Data Language/1). Today, IMS applications and databases can be connected to CICS applications and

□ □ □

Chapter-4

Relational Algebra & Relational Calculus

Q.1 Define the following terms.

Ans.: (i) **Instances and Schemas :**

a) Databases change over time.

b) The information in a database at a particular point in time is called an **Instance** of the database.

c) The overall design of the database is called the database **Schema**.

d) Analogy with programming languages :

- Data type definition - Schema
- Value of a variable - Instance

e) There are several schemas, corresponding to levels of abstraction :

- Physical Schema
- Conceptual Schema
- Sub-Schema (can be many)

(ii) **Data Independence :**

a) The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called **Data Independence**.

b) There are two kinds :

- **Physical Data Independence :**

- The ability to modify the physical scheme without causing application programs to be rewritten
- Modifications at this level are usually to improve performance

- **Logical Data Independence :**
  - The ability to modify the conceptual scheme without causing application programs to be rewritten
  - Usually done when logical structure of database is altered

c) Logical data independence is harder to achieve as the application programs are usually heavily dependent on the logical structure of the data. An analogy is made to abstract data types in programming languages.

(iii) **Data Definition Language (DDL) :**

a) Used to specify a database scheme as a set of definitions expressed in a DDL.

b) DDL statements are compiled, resulting in a set of tables stored in a special file called a data dictionary or data directory.

c) The data directory contains metadata (data about data).

d) The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a **Data Storage and Definition** language

Basic Idea : hide implementation details of the database schemes from the users.

(iv) **Data Manipulation Language (DML) :**

a) **Data Manipulation** is : -

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

b) A DML is a language which enables users to access and manipulate data.

c) The goal is to provide efficient human interaction with the system.

d) There are two types of DML :

- **Procedural** : The user specifies *what* data is needed and *how* to get it.

- **Nonprocedural** : The user only specifies *what* data is needed.
  - Easier for user.
  - May not generate code as efficient as that produced by procedural languages.

e) A **Query Language** is a portion of a DML involving information retrieval only. The terms DML and query language are often used synonymously.

Q.2 What do you mean by Database Manager and explain its responsibilities for DBMS.

Ans.: Database Manager :

a) The **Database Manager** is a **Program Module** which provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

b) Databases typically require lots of storage space (gigabytes). This must be stored on disks. Data is moved between disk and main memory (MM) as needed.

c) The goal of the database system is to **simplify** and **facilitate** access to data. Performance is important. Views provide simplification.

d) **So** the database manager module is responsible for :

- **Interaction with the File Manager :** Storing raw data on disk using the file system usually provided by a conventional operating system. The database manager must translate DML statements into low-level file system commands (for storing, retrieving and updating data in the database).
- **Integrity Enforcement :** Checking that updates in the database do not violate consistency constraints (e.g. no bank account balance below $25).
- **Security Enforcement :** Ensuring that users only have access to information they are permitted to see.
- **Backup and Recovery :** Detecting failures due to power failure, disk crash, software errors, etc., and restoring the database to its state before the failure.
- **Concurrency Control :** Preserving data consistency when there are concurrent users.

e) Some small database systems may miss some of these features, resulting in simpler database managers. (For example, no concurrency is required on a PC running MS-DOS.) These features are necessary on larger systems.

Q.3 Explain the concept of Relational Algebra.

Introduction :

Relational algebras received little attention until the publication of E.F. Codd's relational model of data in 1970. Codd proposed such an algebra as a basis for database query languages. The first query language to be based on Codd's algebra was ISBL, and this pioneering work has been acclaimed by many authorities as having shown the way to make Codd's idea into a useful language. Business System 12 was a short-lived industry-strength relational DBMS that followed the ISBL example. In 1998 Chris Date and Hugh Darwen proposed a language called **Tutorial D** intended for use in teaching relational database theory, and its query language also draws on ISBL's ideas. Rel is an implementation of **Tutorial D**. Even the query language of SQL is loosely based on a relational algebra, though the operands in SQL (tables) are not exactly relations and several useful theorems about the relational algebra do not hold in the SQL counterpart (arguably to the detriment of optimisers and/or users).

Because a relation is interpreted as the extension of some predicate, each operator of a relational algebra has a counterpart in predicate calculus. For example, the natural join is a counterpart of logical AND (). If relations *R* and *S* represent the extensions of predicates *p1* and *p2*, respectively, then the natural join of *R* and *S* (*R S*) is a relation representing the extension of the predicate *p1 p2*.

It is important to realise that Codd's algebra is not in fact complete with respect to first-order logic. Had it been so, certain insurmountable computational difficulties would have arisen for any implementation of it. To overcome these difficulties, he restricted the operands to finite relations only and also proposed restricted support for negation (NOT) and disjunction (OR). Analogous restrictions are found in many other logic-based computer languages. Codd defined the term ***relational completeness*** to refer to a language that is complete with respect to first-order predicate calculus apart from the restrictions he proposed. In practice the restrictions have no adverse effect on the applicability of his relational algebra for database purposes.

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 by 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.

The **Relational Algebra** is a procedural query language.

- Six fundamental operations :

  Unary

  - Select
  - Project
  - Rename

  Binary

  - Cartesian product
  - Union
  - Set-difference
- Several other operations, defined in terms of the fundamental operations :
  - Set-intersection
  - Natural join
  - Division
  - Assignment

- Operations produce a new relation as a result.
- There are some relations which we are using in our queries :

(i) Account

account_number	*branch_name*	*balance*
A-101	*Downtown*	*500*
A-102	*SFU*	*400*
A-201	*Brighton*	*900*
A-215	*Mianus*	*700*
A-217	*Brighton*	*750*
A-222	*Laugheed Mall*	*700*
A-305	*Round Hill*	*350*

(ii) Branch

branch_name	*branch-city*	*assets*
Brighton	*Brooklyn*	*7100000*
Downtown	*Vaneouver*	*9000000*
Mianus	*Horseneck*	*400000*
North Town	*Rye*	*3700000*
SFU	*Burnaby*	*1700000*
Pownal	*Bennington*	*300000*
Lougheed Mall	*Burnaby*	*2100000*
Round Hill	*Horseneck*	*8000000*

(iii) Customer

customer_name	*customer_street*	*customer_city*
Adams	*Spring*	*Pittsfield*
Brooks	*Senator*	*Brooklyn*
Curry	*North*	*Rye*
Glenn	*Sand Hill*	*Woodside*
Green	*Walnut*	*Stamford*
customer_name	*customer_street*	*customer_city*
Hayes	*Main*	*Harrison*
Johnson	*Alma*	*PaloAlto*
Jones	*Main*	*Harrison*
Lindsay	*Park*	*Pittsfield*
Smith	*North*	*Rye*
Turner	*Putnam*	*Stamford*
Williams	*Nassau*	*Princeton*

(iv) Depositor

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

(v) Loan

Loan_number	*branch_name*	*amount*
L-11	*Round Hill*	*900*
L-14	*Downton*	*1500*
L-15	*SFU*	*1500*
L-16	*SFU*	*1300*

L-17	*Downtown*	*1000*
L-23	*Laugheed Mall*	*2000*
L-93	*Mianus*	*500*

(vi) Borrower

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

Fundamental Operations :

(i) **The Select Operation :**

Select selects tuples that satisfy a given predicate. Select is denoted by a lowercase Greek sigma (**σ**), with the predicate appearing as a subscript. The argument relation is given in parentheses following the **σ**.

For example, to select tuples (rows) of the *borrow* relation where the branch is ``SFU'', we would write

$$\sigma_{bname = \text{“SFU”}}(borrow)$$

Let Figure be the *borrow* and *branch* relations in the banking example :

bname	loan#	ename	amount
Downtown	17	Jones	1000
Lougheed Mall	23	Smith	2000
SFU	15	Hayes	1500

bname	assets	bcity
Downtown	9,00,000	Vaneouver
Lougheed Mall	21,000,000	Burnaby
SFU	17,000,000	Burnaby

Figure : The *borrow* and *branch* Relations

The new relation created as the result of this operation consists of one tuple :

(SFU, 15, Hayes, 1500)

We allow comparisons using **=, ≠, <, ≤, > and ≥** in the selection predicate.

We also allow the logical connectives $\vee$ (or) and $\wedge$ (and). For example :

$$\sigma_{bname = \text{"Downtown"} \wedge amount > 1200}(borrow)$$

ename	banker
Hayes	Jones
Johnson	Johnson

Figure : The *client* Relation

Suppose there is one more relation, *client*, shown in Figure, with the scheme

Client_scheme = (ename, banker)

we might write

$$\sigma_{ename = banker}(client)$$

to find clients who have the same name as their banker.

(ii) **The Project Operation :**

Project copies its argument relation for the specified attributes only. Since a relation is a **set**, duplicate rows are eliminated.

Projection is denoted by the Greek capital letter pi (π). The attributes to be copied appear as subscripts.

For example, to obtain a relation showing customers and branches, but ignoring amount and loan#, we write

$$\prod_{bname, ename}(borrow)$$

We can perform these operations on the relations resulting from other operations.

To get the names of customers having the same name as their bankers,

$$\prod_{ename}(\sigma_{ename = banker}(client))$$

Think of **select** as taking rows of a relation, and **project** as taking columns of a relation.

(iii) **The Cartesian Product Operation :**

The **Cartesian Product** of two relations is denoted by a cross (**x**), written

r_1 x r_2 for relations r_1 and r_2

The result of **r_1 x r_2** is a new relation with a tuple for each possible **pairing** of tuples from **r_1** and **r_2**.

In order to avoid ambiguity, the attribute names have attached to them the name of the relation from which they came. If no ambiguity will result, we drop the relation name.

The result **client x customer** is a very large relation. If **r_1** has **n_1** tuples, and **r_2** has **n_2** tuples, then **$r = r_1 x r_2$** will have **$n_1 n_2$** tuples.

The resulting schema is the concatenation of the schemas of **r_1** and **r_2**, with relation names added as mentioned.

To find the clients of banker Johnson and the city in which they live, we need information in both *client* and *customer* relations. We can get this by writing

$\sigma_{banker = "Johnson"}$(client x customer)

However, the *customer. name* column contains customers of bankers other than Johnson. (Why?)

We want rows where *client. name* = *customer. name*. So we can write

$\sigma_{client . ename = customer . ename}(\sigma_{banker = "Johnson"}$(client x customer))

to get just these tuples.

Finally, to get just the customer's name and city, we need a projection :

$\prod_{client . ename , ecity}$ ($\sigma_{client . ename = customer . ename}(\sigma_{banker = "Johnson"}$(client x customer)))

(iv) **The Rename Operation** :

The **Rename Operation** solves the problems that occurs with naming when performing the Cartesian Product of a relation with itself.

Suppose we want to find the names of all the customers who live on the same street and in the same city as Smith.

We can get the street and city of Smith by writing

$$\prod_{\text{street , ecity}} (\sigma_{\text{ename = "Smith"}}(\text{customer}))$$

To find other customers with the same information, we need to reference the *customer* relation again :

$$\sigma_{p}(\text{customer x } (\prod_{\text{street , ecity}} (\sigma_{\text{ename = "Smith"}}(\text{customer}))))$$

where **P** is a selection predicate requiring *street* and *ecity* values to be equal.

Problem: how do we distinguish between the two street values appearing in the Cartesian product, as both come from a *customer* relation?
Solution: use the rename operator, denoted by the Greek letter rho (**ρ**).

We write $\rho_{z}(r)$

to get the relation **r** under the name of **z**.

If we use this to rename one of the two **customer** relations we are using, the ambiguities will disappear.

$$\prod_{\text{customer . ename}} (\sigma_{\text{cust2 . street = customer . street } \wedge \text{ cust2 . ecity = customer . ecity}}$$

$$(\text{customer x } (\prod_{\text{street , ecity}} (\sigma_{\text{ename = "Smith"}}(\rho_{\text{cust2}}(\text{customer})))))$$

(v) **The Union Operation :**

The **Union Operation** is denoted by ∪ as in set theory. It returns the union (set union) of two compatible relations.

For a union operation **r U s** to be legal, we require that

- **r** and **s** must have the same number of attributes.
- The domains of the corresponding attributes must be the same.

To find all customers of the SFU branch, we must find everyone who has a loan or an account or both at the branch.

We need both *borrow* and *deposit* relations for this:

$$\prod_{\text{ename}}(\sigma_{\text{balance = "SFU"}}(\text{borrow})) \cup \prod_{\text{ename}}(\sigma_{\text{bname = "SFU"}}(\text{deposit}))$$

As in all set operations, duplicates are eliminated, giving the relation of Figure.

(a)

ename
Hayes
Adams

(b)

ename
Adams

Figure : The *union* and *set-difference* operations

(i) **The Set Difference Operation**

Set difference is denoted by the minus sign (—). It finds tuples that are in one relation, but not in another.

Thus r — s results in a relation containing tuples that are in **r** but not in **s**

To find customers of the SFU branch who have an account there but no loan, we write

$$\prod_{ename}(\sigma_{balance = \text{"SFU"}}(deposit)) - \prod_{ename}(\sigma_{bname = \text{"SFU"}}(borrow))$$

We can do more with this operation. Suppose we want to find the largest account balance in the bank.

Strategy :

- Find a relation **r** containing the balances **not** the largest.
- Compute the set difference of **r** and the *deposit* relation.

To find **r**, we write

$$\prod_{deposit.balance}(\sigma_{deposit.balance < d.balance}(deposit \times \rho_d(deposit)))$$

This resulting relation contains all balances except the largest one.

Now we can finish our query by taking the set difference:

$$\prod_{balance}(deposit) - \prod_{deposit.balance}(\sigma_{deposit.balance < d.balance}(deposit \times \rho_d(deposit)))$$

Figure shows the result.

(a)

balance
400
500
700

(b)

balance
1300
900

Figure 1 : Find the largest account balance in the bank

Formal Definition of Relational Algebra :

(i) A basic expression consists of either

- A relation in the database.
- A constant relation.

(ii) General expressions are formed out of smaller subexpressions using

- $\sigma_P(E_1)$ select (p a predicate)
- $\prod_S(E_1)$ project (s a list of attributes)
- $\rho_Z(E_1)$ rename (x a relation name)
- $E_1 \cup E_2$ union
- $E_1 - E_2$ set difference
- $E_1 \times E_2$ Cartesian product

Additional Operations :

(i) Additional operations are defined in terms of the fundamental operations. They do not add power to the algebra, but are useful to simplify common queries.

(ii) **The Set Intersection Operation**

Set intersection is denoted by ∩, and returns a relation that contains tuples that are in **both** of its argument relations.

It does not add any power as

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

To find all customers having both a loan and an account at the SFU branch, we write

$$\prod_{ename}(\sigma_{bname = \text{"SFU"}}(borrow)) \cap \prod_{ename}(\sigma_{bname = \text{"SFU"}}(deposit))$$

(iii) **The Natural Join Operation :**

Often we want to simplify queries on a Cartesian product.

For example, to find all customers having a loan at the bank and the cities in which they live, we need *borrow* and *customer* relations :

$$\prod_{borrow.ename,\ ecity}(\sigma_{borrow.ename\ =\ customer.ename}(borrow \times customer))$$

Our selection predicate obtains only those tuples pertaining to only one *cname*.

This type of operation is very common, so we have the **natural join**, denoted by a **x** sign. Natural join combines a cartesian product and a selection into one operation. It performs a selection forcing equality on those attributes that appear in both relation schemas. Duplicates are removed as in all relation operations.

To illustrate, we can rewrite the previous query as

$$\prod_{\text{ename , ecity}} \textbf{(borrow x customer))}$$

The resulting relation is shown in Figure.

ename	ecity
Smith	Burnaby
Hayes	Burnaby
Jones	Vancouver

Figure : Joining *borrow* and *customer* relations

We can now make a more formal definition of natural join.

- Consider **R** and **S** to be **sets** of attributes.
- We denote attributes appearing in **both** relations by **R** ∩ **S**.
- We denote attributes in **either or both** relations by **R** U **S**.
- Consider two relations **r(R)** and **s(S)**.
- The natural join of **r** and **s**, denoted by **r x s** is a relation on scheme **R** U **S**.
- It is a projection onto **R** U **S** of a selection on **r x s** where the predicate requires **r . A = s . A** for each attribute **A** in **R** ∩ **S**.

Formally,

$$\mathbf{r \times s} = \prod_{R \cup S}(\sigma_{r.A1 = s.A1 \wedge r.A2 = s.A2 \wedge \ldots\ldots\ldots \wedge r.An = s.An}\ \mathbf{(r \times s))}$$

where $\mathbf{R \cap S = \{ A_1 , A_2 , \_ \_ \_ , A_n \}}$

To find the assets and names of all branches which have depositors living in Stamford, we need *customer*, *deposit* and *branch* relations :

$$\prod_{\text{bname , assets}}(\sigma_{\text{ecity = "Stamford"}}\ \textbf{(customer x deposit x branch))}$$

Note that **x** is associative.

To find all customers who have both an account and a loan at the SFU branch:

$$\prod_{\text{ename}}\ (\sigma_{\text{bname = "SFU"}}\ \textbf{(borrow x deposit))}$$

This is equivalent to the set intersection version we wrote earlier. We see now that there can be several ways to write a query in the relational algebra.

If two relations **r(R)** and **s(S)** have no attributes in common, then **R** ∩ **S = 0**, and **r x s = s x r.**

(iv) **The Division Operation :**

Division, denoted by ÷, is suited to queries that include the phrase ``for all''.

Suppose we want to find all the customers who have an account at all branches located in Brooklyn.

Strategy: think of it as three steps.

We can obtain the names of all branches located in Brooklyn by

$$r_1 = \prod_{bname} (\sigma_{bcity = \text{“Brooklyn”}} (branch))$$

We can also find all *cname, bname* pairs for which the customer has an account by

$$r_2 = \prod_{ename, bname}(deposit)$$

Now we need to find all customers who appear in r_2 with **every** branch name in r_1.

The divide operation provides exactly those customers:

$$\prod_{ename, bname}(deposit) \div \prod_{bname} (\sigma_{bcity = \text{“Brooklyn”}} (branch))$$

which is simply $r_2 \div r_1$.

Formally,

- Let **r(R)** and **s(S)** be relations.
- Let $S \subseteq R$.
- The relation $r \div s$ is a relation on scheme $R - S$.
- A tuple t is in $r \div s$ if for every tuple t_s in **s** there is a tuple t_r in **r** satisfying both of the following:

  $$t_r[S] = t_s[S]$$

  $$t_r[R - S] = t_s[R - S]$$

- These conditions say that the $R - S$ portion of a tuple t is in $r \div s$ if and only if there are tuples with the $r - s$ portion **and** the **S** portion in **r** for **every** value of the **S** portion in relation **S.**

The division operation can be defined in terms of the fundamental operations.

$$r \div s = \prod_{R-S}(r) - \prod_{R-S}(\prod_{R-S}(r) \times s) - r)$$

(v) **The Assignment Operation**

Sometimes it is useful to be able to write a relational algebra expression in parts using a temporary relation variable (as we did with $\mathbf{r_1}$ and $\mathbf{r_2}$ in the division example).

The assignment operation, denoted ←, works like assignment in a programming language.

We could rewrite our division definition as

$$\mathbf{temp} \leftarrow \prod_{R-S}(\mathbf{r})$$

$$\mathbf{temp} \leftarrow \prod_{R-S}((\mathbf{temp} \times \mathbf{s}) - \mathbf{r})$$

No extra relation is added to the database, but the relation variable created can be used in subsequent expressions. Assignment to a permanent relation would constitute a modification to the database.

Q.4 Explain the term Relational Calculus in DBMS.

Ans.: The Tuple Relational Calculus :

(i) The tuple relational calculus is a nonprocedural language. (The relational algebra was procedural.)

We must provide a formal description of the information desired.

(ii) A query in the tuple relational calculus is expressed as

$$\{ t \mid P(t) \}$$

i.e. the set of tuples **t** for which predicate **P** is true.

(iii) We also use the notation

- **t[a]** to indicate the value of tuple **t** on attribute **a**.
- $t \in r$ to show that tuple **t** is in relation **r**.

Example Queries

For example, to find the branch-name, loan number, customer name and amount for loans over $1200:

$$\{ t \mid t \in \mathbf{borrow} \wedge t[\mathbf{amount}] > 1200 \}$$

This gives us all attributes, but suppose we only want the customer names. (We would use **project** in the algebra.)

We need to write an expression for a relation on scheme *(cname)*.

$$\{ t \mid \exists \delta\ t \in \textbf{borrow}\ (\ t[\textbf{ename}] = s[\textbf{ename}] \ \wedge\ s[\textbf{amount}] > 1200\ )\}$$

In English, we may read this equation as ``the set of all tuples **t** such that there exists a tuple **s** in the relation *borrow* for which the values of **t** and **s** for the *cname* attribute are equal, and the value of **s** for the *amount* attribute is greater than 1200.''

The notation $\exists\ t \in r(Q(t))$ means ``there exists a tuple **t** in relation **r** such that predicate **Q(t)** is true''.

How did we get the above expression? We needed tuples on scheme *cname* such that there were tuples in *borrow* pertaining to that customer name with *amount* attribute **> 1200.**

The tuples **t** get the scheme *cname* implicitly as that is the only attribute **t** is mentioned with.

Let's look at a more complex example.

Find all customers having a loan from the SFU branch, and the the cities in which they live:

$$\{ t \mid \exists \delta\ t \in \textbf{borrow}\ (\ t[\textbf{ename}] = s[\textbf{ename}] \ \wedge\ s[\textbf{amount}] = \text{“SFU”}\ )\}$$

$$\{ \wedge\ \exists\ u \in \textbf{customer}\ (\ u[\textbf{ename}] = s[\textbf{ename}] \ \wedge\ t[\textbf{ecity}] = u[\textbf{ecity}]\ )\}$$

In English, we might read this as ``the set of all *(cname,ccity)* tuples for which *cname* is a borrower at the SFU branch, and *ccity* is the city of *cname*''.

Tuple variable **s** ensures that the customer is a borrower at the SFU branch.

Tuple variable **u** is restricted to pertain to the same customer as **s**, and also ensures that *ccity* is the city of the customer.

The logical connectives $\wedge$(AND) and $\vee$(OR) are allowed, as well as $\neg$(negation).

We also use the existential quantifier $\exists$ and the universal quantifier $\forall$.

Some more examples :

(i) Find all customers having a loan, an account, or both at the SFU branch :

$$\{ t \mid \exists \delta\ t \in \textbf{borrow}\ (\ t[\textbf{ename}] = s[\textbf{ename}] \ \wedge\ s[\textbf{bname}] = \text{“SFU”}\ )\}$$

$$\{ \wedge\ \exists\ u \in \textbf{deposit}\ (\ t[\textbf{ename}] = u[\textbf{ename}] \ \wedge\ u[\textbf{bname}] = \text{“SFU”}\ )\}$$

Note the use of the $\vee$ connective.

As usual, set operations remove all duplicates.

(ii) Find all customers who have **both** a loan and an account at the SFU branch.

Solution: simply change the $\vee$ connective in 1 to a $\wedge$.

(iii) Find customers who have an account, but **not** a loan at the SFU branch.

{ t | ∃ u ∈ deposit (t[ename] = u[ename] ∧ u[bname] = "SFU")}

{ A — ∃ s ∈ borrow (t[ename] = s[ename] ∧ s[bname] = "SFU")}

(iv) Find all customers who have an account at **all** branches located in Brooklyn. (We used **division** in relational algebra.)

For this example we will use implication, denoted by a pointing finger in the text, but by $\Rightarrow$ here.

The formula **P** $\Rightarrow$ **Q** means **P** implies **Q**, or, if **P** is true, then **Q** must be true.

{ t | ∀ u ∈ brach (u[city] = "Brooklyn" ⇒

(∃ s ∈ deposit (t[ename] = s[ename] ∧ u[bname] = s[bname]))}

In English: the set of all *cname* tuples **t** such that for all tuples **u** in the *branch* relation, if the value of **u** on attribute *bcity* is Brooklyn, then the customer has an account at the branch whose name appears in the *bname* attribute of **u**.

Formal Definitions :

(i) A tuple relational calculus expression is of the form

{ t | P(t) }

where **P** is a **formula**.

Several tuple variables may appear in a formula.

(ii) A tuple variable is said to be a **free variable** unless it is quantified by a $\exists$ or $\forall$. Then it is said to be a **bound variable**.

(iii) A formula is built of **atoms**. An atom is one of the following forms:

- **s ∈ r**, where s is a tuple variable, and r is a relation ($\notin$ is not allowed).
- **s[x]** θ **u[y]**, where s and u are tuple variables, and **x** and **y** are attributes, and θ is a comparison operator **(<, ≤, =, ≠, >, ≥).**
- **s[x]** θ **c,** where c is a constant in the domain of attribute **x**.

(iv) **Formulae** are built up from atoms using the following rules :

- An atom is a formula.
- If **P** is a formula, then so are $\neg$ **P and (P)**.
- If P_1 and P_2 and are formulae, then so are $P_1 \wedge P_2$, $P_1 \vee P_2$ and $P_1 \Rightarrow P_2$.
- If **P(s)** is a formula containing a free tuple variable **s**, then

  $\exists$ **s** $\in$ **r(P(s)) and** $\forall$ **s** $\in$ **r(P(s))** are formulae also.

(v) Note some equivalences : -

- $P_1 \wedge P_2 = \neg (\neg P_1 \vee \neg P_2)$
- $\forall t \in r(P(t)) = \exists t \in r(\neg P(t))$
- $P_1 => P_2 = \neg P_1 \vee P_2$

The Domain Relational Calculus :

Domain variables take on values from an attribute's domain, rather than values for an entire tuple. Formal Definitions

(i) An expression is of the form

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n) \}$$

where the x_{i1} $1 \leq i \leq n_1$ represent domain variables, and **P** is a **formula**.

(ii) An atom in the domain relational calculus is of the following forms

- $< x_1, x_2, \ldots, x_n > \in r$ where **r** is a relation on **n** attributes, and x_{i1} $1 \leq i \leq n_1$, are domain variables or constants.
- **x** θ **y,** where **x** and **y** are domain variables, and θ is a comparison operator.
- **x** θ **c**, where c is a constant.

(iii) **Formulae** are built up from atoms using the following rules :

- An atom is a formula.
- If **P** is a formula, then so are $\neg$**P** and **(P)**.
- If P_1 and P_2 are formulae, then so are $P_1 \wedge P_2$, $P_1 \vee P_2$ and $P_1 \Rightarrow P_2$.
- If **P(x)** is a formula where **x** is a domain variable, then so are $\exists$**x (P(x))** and $\forall$ **x P(x))** .

Example Queries :

(i) Find branch name, loan number, customer name and amount for loans of over $1200.

$$\{ < b_1\ I_1\ c_1\ a > \mid b_1\ I_1\ c_1\ a \in borrow \wedge a > 1200 \}$$

(ii) Find all customers who have a loan for an amount > than $1200.

$$\{ < c > \mid \exists\ b_1\ I_1\ a\ (< b_1\ I_1\ c_1\ a > \in borrow \wedge a > 1200)\}$$

(iv) Find all customers having a loan from the SFU branch, and the city in which they live.

$$\{ < c_1\ x > \mid \exists\ b_1\ I_1\ a\ (< b_1\ I_1\ c_1\ a > \in borrow$$

$$\wedge\ b = \text{“SFU”}\ > \wedge\ \exists\ y\ (\ c_1\ y_1\ x > \in customer\ ))\}$$

(v) Find all customers having a loan, an account or both at the SFU branch.

$$\{ < c > \mid \exists\ b1\ I1\ a\ (< b1\ I1\ c1\ a > \in borrow \wedge b = \text{“SFU”})$$

$$V\ \exists\ b1\ a1\ n\ (< b1\ a1\ c1\ a > \in deposit \wedge b = \text{“SFU”})\}$$

(vi) Find all customers who have an account at **all** branches located in Brooklyn.

$$\{ < c > \mid \forall\ x1\ y1\ z\ (\ \neg\ (\ < x1\ y1\ z > \in branch)$$

$$\vee\ z \neq \text{”Brooklyn”}\ b1\ a1\ n \vee (\exists\ a1\ n\ (< x1\ a1\ c1\ n > \in deposit\ )))\}$$

If you find this example difficult to understand, try rewriting this expression using implication, as in the tuple relational calculus example. Here's my attempt:

$$\{ < cn > \mid \forall\ bn1\ as1\ bc\ (\ (\ < bn1\ as1\ bc > \in branch)$$

$$\vee\ bc \neq \text{”Brooklyn”}) \Rightarrow \exists\ an1\ ba\ (< cn1\ an1\ ba1\ bn > \in deposit\ )))\}$$

I've used two letter variable names to get away from the problem of having to remember what **x** stands for.

□ □ □