University Mohamed Boudiaf of M'sla
Faculty of Mathematics and Computer Science
Department of Computer Science

ADVANCED DATABASES

Dr. Rabah Mokhtari

2023 – Version 1.0

CONTENT

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Today, databases and database systems are used on almost all organization computer systems including business, electronic commerce, engineering, medicine, genetics, and education. Initially, we may cite a general definition of what a database is? "A database is a collection of related data" [9]. This collection of data is stored for later retrieval and processing[16]. Data is used to reference known facts like, for example, person names, phone numbers and product prices that can be recorded or stored in physical drives and may be described by logical meaner. By related data, we mean that a random assortment of data cannot be qualified as a database. But the term database is usually restricted with the following properties [9]:

- A database represents some aspect of the real world, and the set of facts represented in a database is sometimes called the Universe of Discourse (**UoD**).
- A database is a logically coherent collection of data with some inherent meaning.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

1.1 FUNDAMENTAL CONCEPTS AND FEATURES

1.1.1 *Data Vs Information*

- **Data**

  Data is collection of raw (or unorganized) facts. It simply exists and has no significance beyond its existence. It can exist in any form, usable or not. It is always need to be processed [16]. Raw data are typically the results of measurements and can be the observation of a set of variables. For example, age of students, the height of students, blood group of students, are generally considered as "data".

- **Information**

  Valuable or useful data is called information. After processing, organization and presenting data in a given situation so to make it useful, it becomes Information. For example, height of a particular student may be considered as "information"[16].

1.1.2 *Non-database Vs Database-Management Systems*

Before the advent of database management, data was stored in data file. A non-database (or **file-based**) system is a program or a collection of programs responsible for data file managing and processing. While a Database Management System (DBMS) is a collection of programs that enable to create and maintain a **database**. Section 1.1.6 presents some advantages of DBMSs over the file-based systems.

1.1.3 *Characteristics of the Database Approach*

- Data Abstraction: A data model is used to hide storage details and present the users with a conceptual view of the database.
- Support of multiple views of the data: the database is populated for several users using different views according to the interest of each user.
- Concurrency : Allowing a set of concurrent users to retrieve from and to update the database.

1.1.4 *Database Management System (DBMS)*

Database Management System (DBMS) is a software system that is used to define, create, manipulate, and maintain databases[16]. Through a DBMS, users may interact with and manage the database. Popular DBMSs include Microsoft Access, Oracle, DB2, MySQL, and Microsoft SQL Server.

1.1.5 *Classification of DBMS users*

A DBMS user is anyone who directly or indirectly uses the DBMS. The DBMS users operate in different level according to their appropriate roles. Data user often refers to:

1. **Database Administrator (DBA)**

   The DBA is the person or the group of person in charge of performing database administration tasks to manage both the database and the use of the DBMS. The DBA has full authority and privileges allowed by the DBMS and can assign and remove privileges to and from other users. He is also responsible for the DBMS maintenance and of the database management and security.

2. **Database Designer**

   The database designer is responsible for defining the detailed database design, including data structure, relationships between data, constraints and other database specific constructs needed to the database creation.

3. **End Users**

   End Users are the people who interact directly with the database through utilities. They are authorized to query and update all or part of the database using standards of Queries to respond quickly to other user requests.

4. **Application Programmers**

To access and eventually change data, an application programmer interact with the database from programs written in high-level programming languages such as C++ and Java.

1.1.6 *Advantages of Database Approach*

- **Getting more information from the same amount of data –** The primary goal of a data computer system is to turn data (recorded facts) into information (the knowledge gained by processing those facts). In non-database, or file-based system, this task can be extremely difficult to fulfill. In other cases is impossible. Given the power of a DBMS, the information is available, and the process of getting it is quick and easy[14].
- **Data integrity and security –** DBMS can provide security to databases by assigning privileges to different users. It provides authorization or access controls to different classes of users to perform different operations on databases, such as creation, modification, deletion and updation of data[16]. The **integrity constraints** is a rule that data must follow in the database[14]. For example, the "department number" given for any "employee" must be one that is already in the database.
- **Controlling Data Redundancy and Inconsistency –** In file-based system, every user group maintains its own data files. This may lead to duplicate data in different files. Several problems may occur since duplicated data is stored, most importantly the wastage of storage space and inconsistency errors that come with updating the duplicated data.

  System developers and database designers often use data normalization to minimize data redundancy and inconsistency [16].
- **Data independence –** Application programs should be as independent from details of data storage and representation. DBMS provides us abstract view of the data to insulate application code from such details[16].
- **Increasing programmer productivity –** A DBMS frees the programmers who are writing database access programs from having to engage in mundane data manipulation activities, such as adding new data and deleting existing data, thus making the programmers more productive[14].

1.1.7 *Disadvantages of Database Approach*

- **Cost of stuff training** The supply and operation of a DBMS is often very complex and quite costly so hiring and training of users at all level is required. So a lot of money has to be paid to run the DBMS[16].
- **Cost of Hardware & Software** A computer with a high speed of data processing and a large storage capacity is required to run the DBMS software hosting the database. Similarly, DBMS software may be also very costly[16].

- **Cost of Data Conversion** When a file-based system is replaced with a database system, the conversion of the data stored into data file from file-based system to database file is very difficult and costly method. You have to hire database and system designers along with application programmers[16].
- **More difficult recovery** When the database is being updated by many users at the same time. The database must first be restored to the condition it was in when it was last known to be correct; any updates made by users since that time must be redone. The greater the number of users involved in updating the database, the more complicated this task becomes[14].

1.2 HISTORY OF DATABASE MANAGEMENT SYSTEMS

To gain insight into the present state of database models and architectures, we need to present a historical introduction of the evolution of database management systems. This evolution has been influenced by the increasing demand of organizations for new database architectures.

A good historical presentation of evolution of Data Processing and Database Management Systems may be found in [2]. We will see that each new database architecture had to add new concepts and features to treat problems occurred in previous architectures.

1.2.1 *Flat file processing*

The early computers were very large and cumbersome to maintain, but they were perfect for performing repetitive tasks as payroll calculations, and organizations soon began to see that high volume for repetitive data storage and processing tasks such as payroll were ideal applications for these computers. In that time, many of "database" systems were really nothing more than a loosely coupled collection of files. [2].

These were called "flat files" because data was stored as fixed length records in flat-files in a **linear fashion**, such that it was necessary to read the file from front-to-end until retrieving the desired record and there are no structures for indexing or recognizing relationships between records because flat-file systems were bound by their linear nature. Flat files are said to be non-keyed files because records was always retrieved in the same order.

An access method defines the technique that is used to store and retrieve data. The terms QSAM and BSAM standing respectively for *Queued Sequential Access Method* (pronounced "quesam") and *Basic Sequential Access Method* (pronounced "bee-sam") were often used to describe physical sequential files in IBM mainframe environment[2].

1.2.2 *Basic Direct Access Method (DBAM)*

As more and more data was stored on disks, organizations struggled to bypass the linear nature of flat file organization. When storing records on disk, each block can be identified

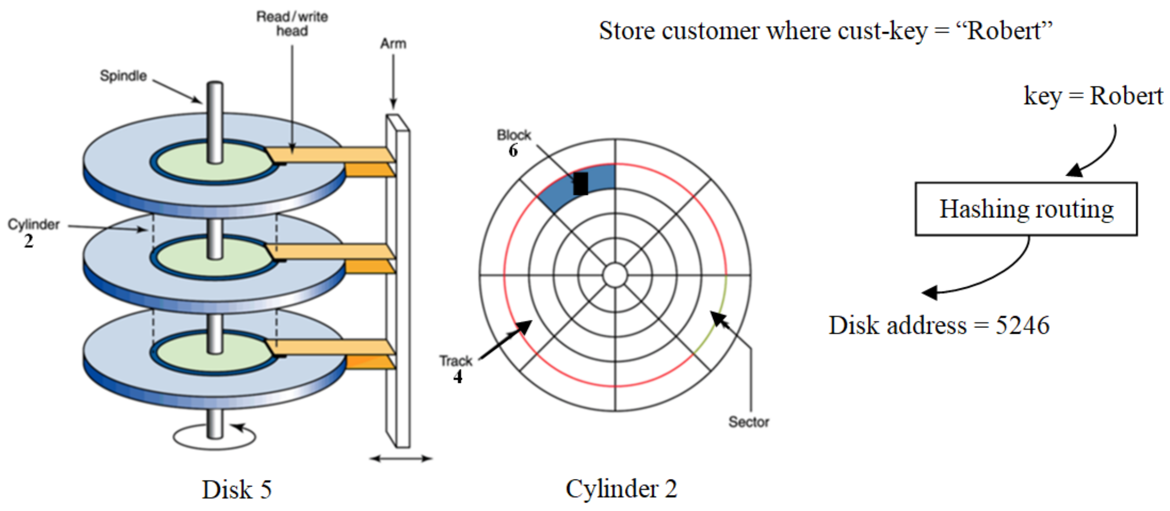


Figure 1: Hashed file storage

by a unique disk address. When we know the address of a record in disk, it can be retrieved very quickly.

Unlike, sequential access methods used in flat file systems, the BDAM (pronounced "bee-damn") method uses hashing algorithm to determine the disk address of the target stored record[2]. Each record is identified by a unique symbolic key. Using the record symbolic key value, the BDAM algorithm will compute the target location (address) of the record. Since we can go directly to the record, BDAM provides much faster access and retrieval of records. A direct access file is sometimes called a "keyed" file because the key is used to generate the disk address (see Figure 1). A disk address includes the disk number, the cylinder address, the track, and the block address.

1.2.3 *The Indexed Sequential Access Method (ISAM)*

While the BDAM method provides presents a good solution for fast storage and retrieval of data, the high cost of disk storage made it a very expensive proposition. However, new methods like ISAM (Indexed Sequential Access Method) and VSAM (Virtual Sequential Access Method) used flat files with indexes to speed data retrieval. These new methods became a very popular alternative to BDAM.

To understand indexing, let's take an example from a book. Just as you use an index in a book to find what you want quickly, a computer index can speed the retrieval of information. In the simplest index structures, the index only contains two fields. One field is the symbolic key and the second field contains the disk address of record that contains the key value. In most file management systems, the index file kept as completely separate file from the master file. When a record is requested, based upon an index key, the program will scan the index, locate the symbolic key, and then retrieve the record from the file based upon the location that was specified in the index column[2].

In this fashion (see Figure 2), a flat file can be logically recognized to retrieve record in any desired order, regardless of the physical sequencing of the data. As new records are

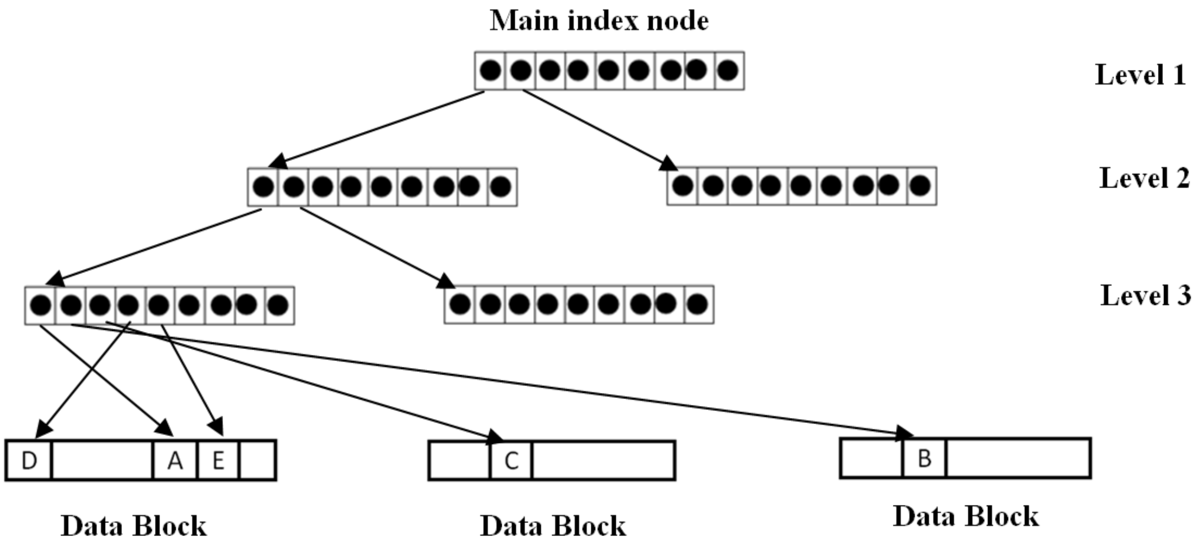


Figure 2: A simple index retrieval

added to the end of the master table, the ISAM file system will automatically adjust the indexes to account for the change.

ISAM was developed at IBM in late 1960s[6]. Like physical sequential files, ISAM stores the records back-to-back, making for very efficient use of disk space, with only the Inter-Block Gap (IBG[1]) between records. However, unlike physical sequential format which may be stored on tape, ISAM files must be stored on disk since addresses are needed to create the indexes and the physical sequence of records within ISAM is not important since the indexes take care of the access to the records[2].

A single ISAM file may have dozens of indexes, each allowing the files to be retrieved in some predefined order. In some cases, the size of the indexes will exceed the size of the master file, but this is still less expensive than the disk wastage that occurs with BDAM files[2]. The main advantages of the ISAM organization are its simplicity, small space overhead and fast query time[6].

Another popular file-access method was introduced by IBM, called the *Virtual Storage Access Method* or VSAM. VSAM, like its cousin ISAM, allows for physical sequencing files to be indexed on multiple data items. By having multiple indexes, data can be retrieved directly in several ways and you can access data anywhere in the file using different index[2].

1.2.4 *Shortcomings of Flat Files*

Needless to say, there were many problems and difficulties with flat-file database systems, such as:

- **Data sharing –** Before the invention of centralized computer resources, each department within an organization would develop their own system, usually implementing their own unique file structures and programming language. Because of

[1]IBG is a break between data records on hard drive and magnetic tape to prevent data overwrites

this, “islands of information” sprang-up within companies and it was very difficult for departments to share information.

- **Duplication of data –** Departments within a company would often duplicate another department’s data, leading to higher disk storage costs. The updating of duplicated data may lead to inconsistency errors.

- **Data file structure modification –** If a data file structure ever changed, correcting all of the programs that needed modification was almost impossible.

- **Backup and recovery –** Flat files possessed no real backup system nor recovery methods. Programmers had to write programs to backup a system before updates started. If a failure occurred at any times during the update process, the files were corrupted should be restored from the backup and rerun the update from the beginning.

1.2.5 *The Era of formal Database Management*

Shortcomings of flat-file systems prompted the development of new approach in managing of amounts of organizational data called *database approach*.

Prior to database management, data resided in vastly different internal formats, and data was not managed in consistent fashion. Even more onerous, the pre-database systems were extremely difficult to communicate with other data systems. It is important to remember that a database management system provides far more than just a uniform repository for information. All database management systems provide the following features:

1. Recovery of incomplete transactions (rollback)

2. A mechanism for recovering transactions after disk failure (roll forward)

3. Internal tools for management of relationships between data.

4. Locking and concurrency tools for simultaneous multiple user access.

5. A common access language that can be embedded in procedural code.

The database management systems of 1990s also provide:

1. Internal support for maintaining business rules (referential integrity).

2. Read consistency for long-running queries.

3. Support for distributed updates (two-phase commit).

4. Attaching behaviors to data (methods).

The early Database Managers access methods were based on BDAM or VSAM and indexes were often created to speed-up data access. In the late 1950s, IBM developed a prototype computer database to demonstrate that data could be stored, retrieved, and updated in a structured format. This database became known as the Information Management System, or IMS.

IMS was a revolutionary idea since it allows data access by numerous programs in different languages and was designed to support the multiuser needs of larger organizations. Even more important, the creation of IMS codified the industries belief that data was important, and needed to be managed and controlled in a consistent fashion.

1.2.6 *The Hierarchical Database Model*

As we noted, the major difference between flat-file systems and BDAM is the ability to store both data as well as the relationships between the data. The hierarchical database model was first introduced as IMS (Information Management System), which was released in 1960. The hierarchical database model used pointers to logically link related data items, and it does this with the use of "child" and "twin" pointers. In 1998, IMS still enjoys a large following among users with large databases and high-volume transaction requirements. A hierarchical database is very well suited to modeling relationships that are naturally hierarchical. For example, within an organization, we see that each executive has many managers, each manager has many supervisors, and each supervisor has many workers. Basically, a hierarchy is a method of organizing data into descending one-to many relationships, with each level in the hierarchy having a higher precedence than those below it.

A hierarchy is just an arrangement of structures called nodes, and the nodes are connected by lines or "branches". You can think of these lines or branches as a connection to the next level of more specific information. The highest node is called the root node, and queries must through this node on their way down the hierarchy. In our example (Figure 3), UNIVERSITY is the root node. Every node, except the root node, is connected upward to only one "parent" node. Nodes have a parent-child relationship, and a parent node is directly above the child node. We also see that the node called COLLEGE OF ENGINEERING is parent of ELECTRICAL ENGINEERING. Since a child node is always one level directly below its parent node, the ELECTRICAL ENGINEERING node is a child of COLLEGE OF ENGINEERING node. Note that a parent node can have more than one child node, but a child node may only have one parent.

When we talk about a hierarchical database, the nodes we talked about become "segment types". A segment type is simply user-defined category of data, and each segment contains fields. Each segment has a key field, and the key field is used to retrieve the data from the segment. There can be one or more fields in segment, and most segment also contain multiple search fields.

Expanding on Figure 3, let's add some data fields to our UNIVERSITY segment. Let's begin by describing the data for each UNIVERSITY. UNIVERSITY information might include the UNIVERSITY name, the mailing address, and phone number. IMS is well-suited for modeling systems in which the entities (segment types) are composed of descending one-to-many relationships. Relationships are established with "child" and "twin" pointers, and these pointers are embedded into the prefix of every record in the database.

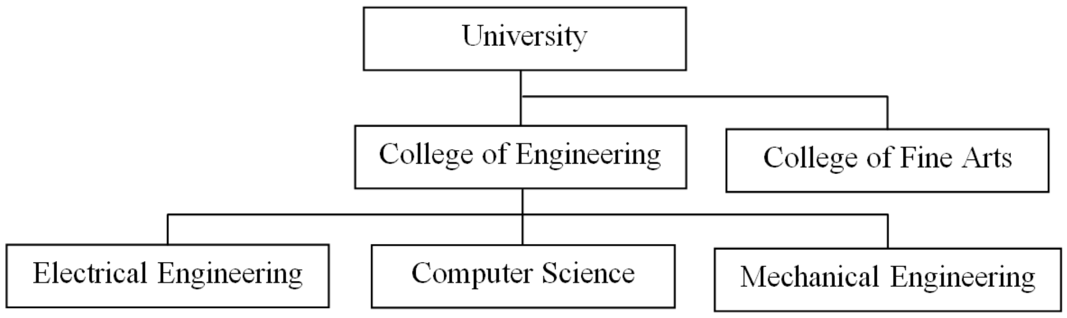


Figure 3: A simple hierarchical chart

Referring to the Figure 3, we see six segments:

1. UNIVERSITY
2. College of Engineering
3. College of Fine Arts
4. Electrical Engineering
5. Computer Science
6. Mechanical Engineering

We also have four hierarchical paths:

1. University, College of Engineering, Electrical Engineering
2. University, College of Engineering, Computer Science
3. University, College of Engineering, Mechanical Engineering
4. University, College of Fine Arts

A hierarchical path defines the access method, and the path is like an imaginary line that begins at the root segment and passes through segment types until it reaches the segment type at the bottom of the inverted tree. One advantage to a hierarchical database is that if you only wanted information on COLLEGES, the program would only have to know the format and access the store segment. You would not have to know that any of other segments even exist, what their fields are, or what relationship exists between the segments.

Hierarchical databases have rigid rules in relationships and data access. For example, all segments have to be accessed through the parent segment. The exception to this is, of course, the root segment because it has no parent.

The IMS database has concurrent control, and a full backup and recovery mechanism. The backup and recovery protects the system from a failure of IMS itself, an application program, a database failure, and an operating system failure. The recovery mechanism for application programs stores "before" and "after" images of each record which was changed, and these images could be used to "roll-back" the database if a transaction failed to complete.

If there was a disk failure the image could be "rolled-forward". IMS was used with the CICS[1] (Customer Information Control System) teleprocessing monitor to develop the first on-line database systems for the mainframe.

Three main advantages of hierarchical databases are: (1) a large base with a proven technology that has been around for decades, (2) the ease of using a hierarchy or tree structure, and (3) the speed of the system (exceeding 2000 transactions per second). Some disadvantages of hierarchical databases are because of rigid rules in relationships, insertion and deletion can become very complex, access to child segment can only be done through the parent segment (start at the root segment). While IMS is very good modeling hierarchical database relationships, complex data relationships such as many-to-many and recursive many-to-many, like BOM (Bill-Of-Material) relationships, had to be implemented in very clumsy fashion, by using "phantom" records. The IMS database also suffered from its complexity. To become proficient in IMS you need months of training and experience. As result, IMS development remains very slow and cumbersome.

1.2.7 *The CODASYL Network Model (known as Navigational Model)*

During the 1960s, several major BDAM products were created using the CODASYL[2] Network Database Management System (DBMS) specifications developed by the Conference on Data Systems Language (CODASYL). Also involved were two subgroups of CODASYL: the Database Task Group (DBTG) and the Data Description Language Committee (DDLC).

CODASYL and its subgroups are an organization of volunteer representatives of computer manufacturers and users. While CODASYL began in 1959, the first set of DBMS specifications were not produced until 1969. This set of specifications was revised, and the first real CODASYL DBTG specifications of the CODASYL standard approach were in 1971.

The CODASYL approach was a very complicated system and required substantial training. It depended on a "manual" navigation technique using a linked data set, which formed a large network. Searching for records could be accomplished by one of three techniques:

1. Using the primary key (also known as the CALC key)
2. Moving relationships (also called sets) to one record from another
3. Scanning all records in sequential order

Basically, all database management systems share some features with CODASYL DBTG specifications. From the early CODASYL DBTG specs, the data model was called a "Network" data model, and the model that CODASYL DBTG developed became the basis

[1]CICS is a family of mixed language application servers that provide online transaction management and connectivity for applications on IBM mainframe systems under z/OS and z/VSE operating systems.

[2]CODASYL was also responsible for designing COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) language in 1959.

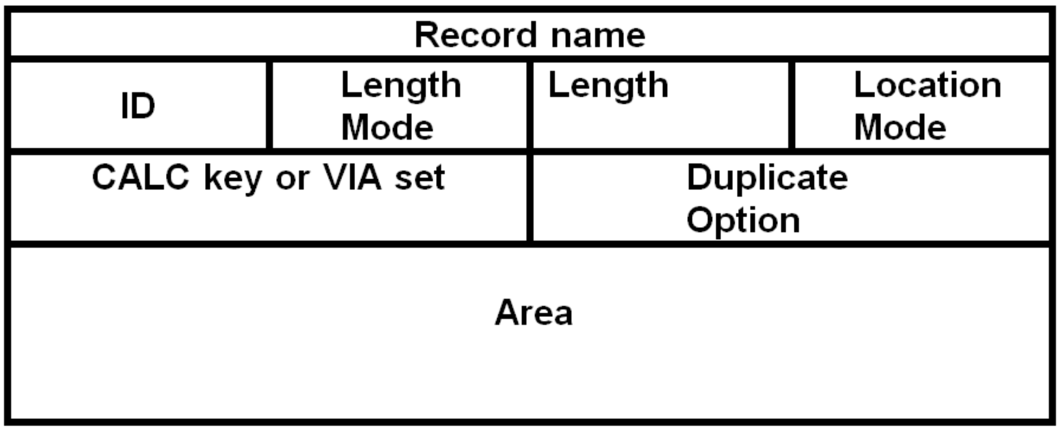


Figure 4: Bachman diagram layout of record structure

for new database systems like IDMS (Integrated Database Management System) from Cullinet in 1970.

The Data Base Task Group (DBTG) CODASYL Specifications included Schema definition, Device Media Control Language (DMCL) definition, Data Manipulation Language (DML) definition. It also included the concept of a database "area" which referred to the physical structure of the data files. The Logical Structure of the database was defined by a Data Definition Language (DDL), and a user view of the data was defined by a subschema. The DML commands were used to navigate through the linked-list structures hat comprised the database. The CODAYL DML verbs included FIND, GET, STORE, MODIFY, and DELETE. The Data Base Administrator (BDA) functions in a CODASYL database included: data structure or schema, data integrity, security, and authorization. Also a Data Base Manager (DBM) function was defined which included: operation, backup/recovery, performance, statistics, auditing.

The CODASYL model used two storage methods, DBAM and linked-list data structure, and DBAM used a hashing algorithm to store and retrieve records.

Because of the many choices that can be made in the design of a Network database, it is important to review the design with as many people as possible. Charles W. Bachman[1] developed a "diagram" that represented the data structures as required by CODASYL. This diagram method becomes known as the Bachman diagram (Figure 4).

The Bachman diagram describes the physical constructs of all record types in the database. The rectangles of Bachman diagram are subdivided into four rows. The top row of the box contains the record name. Row two contains the numeric identification ID number (each record is given a number which is associated with the record name), the length mode which is fixed or variable, the length of records, and the location mode (CALC, or VIA). Row three contains for CALC, the field serving as the CALC key, and for VIA SET, the set name. Row four contains the era designated.

[1]Charles Bachman is best known for his invention of the first random access database management system, the Integrated Data Store (IDS) and he won the ACM's 1973 A.M. Turing Award for his outstanding contributions to database technology

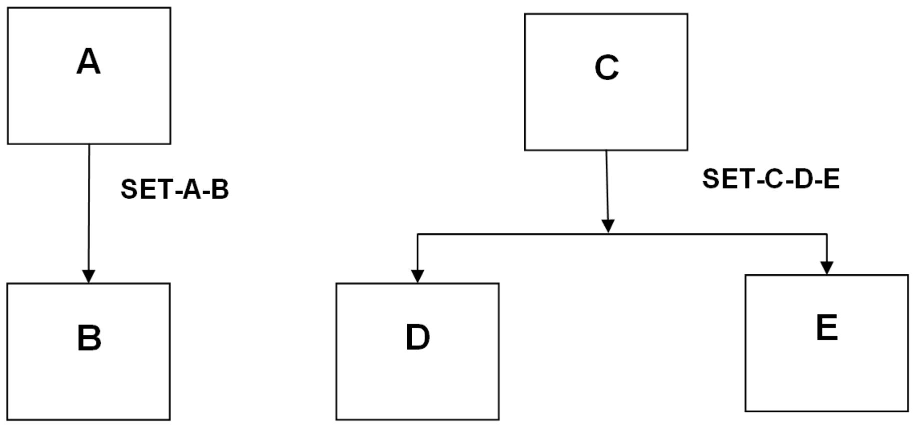


Figure 5: A set type of one owner record type and one or more member record types

The set type is shown by Bachman arrow pointing from the owner record type to the member record type (see Figure 5).

Set name is the owner name hyphen member name. Pointers are Next, Prior and Owner option is used for insertion and retention (MA, OA, MM, OM); the set order is (First, Last, Next, Prior, or Sorted); and the mode is (Chain or Index).

SHORTCOMINGS OF CODASYL NETWORK MODEL

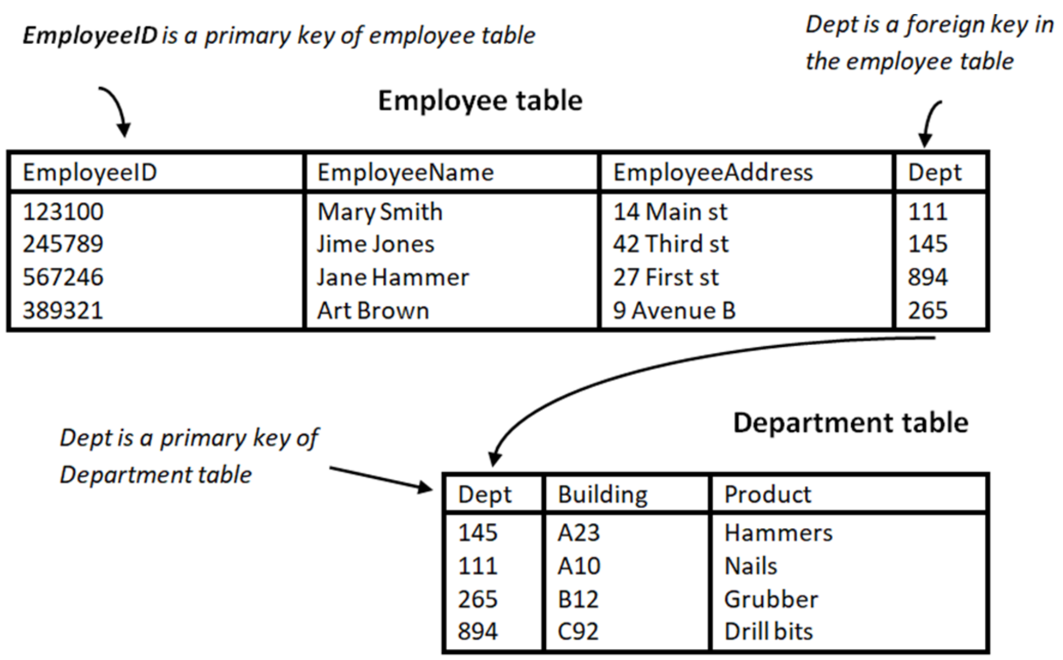
The design of the CODASYL network model was very complex, and consequently very difficult to use. Network databases, very much like hierarchical databases, are very difficult to navigate. The Data Manipulation Language (DML), designed for complex navigation, was a skill that required months of training.

Implementing structural changes was extremely difficult with network databases since data relationships are "hard-linked" with embedded pointers. Adding an index or new relationship requires special utility programs that will "sweep" each and every record in the database. As records are located, the prefix is structured to accommodate the new pointers. Object-oriented databases will encounter this same problem if a class hierarchy needs to be modified.

Eventually, the CODASYL approach lost its popularity as simpler, easier-to-work-with systems came on the market.

1.2.8 *The Relational Database Model*

Edgar Codd worked at IBM in San Jose, California, in one of their offshoot offices that was primarily involved in the development of hard disk systems. He was unhappy with the navigational model of the CODASYL approach and the IMS model. He wrote a series of papers, in 1970, outlining novel ways to construct databases. His ideas eventually evolved into a paper titled, *A Relational Model of Data for Large Shared Data Banks*[3], which described new method for storing data and processing large databases. Records would not be stored in a free-form list of linked records, as in CODASYL navigational model,


EmployeeID is a primary key of employee table
Dept is a foreign key in the employee table
Employee table
Dept is a primary key of Department table
Department table

EmployeeID	EmployeeName	EmployeeAddress	Dept
123100	Mary Smith	14 Main st	111
245789	Jime Jones	42 Third st	145
567246	Jane Hammer	27 First st	894
389321	Art Brown	9 Avenue B	265

Dept	Building	Product
145	A23	Hammers
111	A10	Nails
265	B12	Grubber
894	C92	Drill bits

Figure 6: A simple relational chart

but instead used “tables” (also referred to as “entities”). Each table has "rows" (also referred to as “records” or “tuples”) and “columns” (also referred to as "attributes"). In relational database model any tables can be linked together. The relational model is based on a collection of mathematical principles drawn primarily from set theory and predicate logic. These principles were first applied to the field of data modeling in the late 1960s by Dr. Edgar Codd. Tables basically correspond to segment type in hierarchical and record types in the network models. Relational tables are independent, unlike hierarchical and network models that are pointer connected, and can contain only one type of record, and each record has a fixed number of fields that are explicitly named. A table will always have a field or several fields that make a unique identifier called “primary key”. Data redundancy is reduced via a process called normalization.

A primary key uniquely identifies a row in a table and a "foreign key" allows you to join two or more tables together by using a primary key in one table with a non key field in another table (see Figure 6).

Relational databases made the following improvement over hierarchical and network databases:

1. **Simplicity**: The concept of tables with rows and columns is extremely simple and easy to understand. End users have a simple data model. Complex network diagrams used with hierarchical and network databases are not used with a relational database.

2. **Data independence**: Data independence is ability to modify data structure (in this, case, tables) without affecting existing programs. Much of this is because tables are not hard linked to one another. Columns can be added to tables, tables can be added to the database, and new data relationships can be added with little or no

restructuring of the tables. A relational database provides a much higher degree of data independence than do hierarchical and network databases.

3. **Declarative data access**: One of the best improvements of the relational model over its predecessors was its simplicity. Rather than having dozens of DML commands, the relational model introduced a declarative language called Structured Query Language (SQL), also known as "sequel", to simplify data access and manipulation. The SQL user specifies what data they want, and then the embedded SQL, a procedural language determines how to get the data. In relational data access, the user tells the systems the conditions for the retrieval of data. The system then gets the data that meets the selection conditions in the SQL statement. The database navigation is hidden from the end user or the programmer, unlike a CODASYL navigation DML language, where the programmer had to know the details of the access path.

1.2.9 *The Object-Oriented Database Model*

The term of Object-Oriented, abbreviated OO, has its origins in OO programming language. The main idea the OO paradigm is to couple the data with behavior. The first procedural language to do this task is Simula language, which was proposed in the late 1960s [9] and used in operations research tasks to simulate the behavior of entities. Object in data management sense was first developed by Xerox Corporation's Palo Research Center (PARC) in the early 1970s [2]. In 1970, Xerox PARC developed the programming language Smalltalk, which was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a pure OO programming language, meaning that it was explicitly designed to be object-oriented.

OO databases management approach emphasizes a more natural representation of the data. In late 1990s environments, the data models are more demanding. They need to handle audio, video, text, graphics, etc. These requirements demand a more flexible storage format than hierarchical, network and relational databases can provide. Only OO databases will be able to support this kind of demand [2].

1.3 conclusion

In this introductory chapter, we discussed the fundamental concepts and features of database management systems (DBMSs), which are software applications that are used to store, organize, and manage data. We examined the differences between non-database and database management systems and the advantages that DBMSs provide, such as data consistency, data integrity, and data security.

We also provided a brief history of DBMSs, which began with the flat file processing systems of the 1960s, followed by the hierarchical and CODASYL databases of the 1970s. We discussed how these early DBMSs evolved into the relational databases of the 1980s, which are still widely used today.

Throughout the chapter, we introduced key concepts and terminology, including data modeling, database schema, database instances, and transactions. We also discussed the role of database administrators and the importance of database performance tuning.

In conclusion, this chapter provides a foundational understanding of DBMSs, including their history, basic concepts, and features. By understanding the principles of database management, readers will be better equipped to design, implement, and manage databases effectively, which are essential for businesses to store and manage their data efficiently and securely.

2 RELATIONAL DATABASE MODEL

The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a paper titled, *A Relational Model of Data for Large Shared Data Banks* [3], and it attracted immediate attention due to its simplicity and mathematical foundation. One of the major advantage of the relational model is its uniformity. All data is viewed as stored in tables, with each row in the table having the same format. The model uses the concept of a *mathematical relation* and has its theoretical basis in set theory and first-order predicate logic [9].

This chapter presents the main concepts of the model, and relational algebra and relational calculus which are associated with the relational model.

2.1 MANUFACTURING COMPANY DATABASE EXAMPLE

Almost illustration examples used in the rest of this chapter are taken from an example of database of a manufacturing company inspired by Date, C.J. [4] (see Section 2.1).

We consider the database of an example of manufacturing company called **KnowWare Inc.** in a little detail. Such an enterprise will typically wish to record information about the *projects* it has on hand; the *parts* that are used in those projects; the *suppliers* who are under contract to supply those projects; the *warehouse* in which those parts are stored; the *employees* who work on those projects; and so on. Projects, parts, suppliers, and so on, thus constitute the basic entities about which KnowWare Inc. needs to record information. Refer to Figure 7.

2.2 FORMALIZATION OF RELATIONS

2.2.1 *Relation scheme, Relation and Tuple*

A *relation scheme* R is a finite set of *attribute names* $\{A_1, A_2, ..., A_n\}$. Corresponding to each attribute name A_i is a set $D_i, 1 \leqslant i \leqslant n$, called the *domain* of A_i. We also donate the domain of A_i by $dom(A_i)$. Attribute names are sometimes called *attribute symbols* or simply *attributes*. The domains are arbitrary, non-empty set, finite, or countably infinite. Let $D = D_1 \cup D_2 \cup ... \cup D_n$.

A *relation* (or **relation state**) r on relation scheme R, denoted by $r(R)$, is a finite set of mappings $\{t_1, t_2, ..., t_p\}$ from R to D with the restriction that for each mapping $t \in r, t(A_i)$ must be in $D_i, 1 \leqslant i \leqslant n$. The mappings are called *n-tuples* [12]. Each **n-tuple** is an ordered list of n values $t =< v_1, v_2, ..., v_n >$, where each value $v_i, 1 \leqslant i \leqslant n$, is an element of $dom(A_i)$ or is a special NULL value [9]. The tuples of a relation r are all distinct, so one tuple can not be duplicated in the same relation.

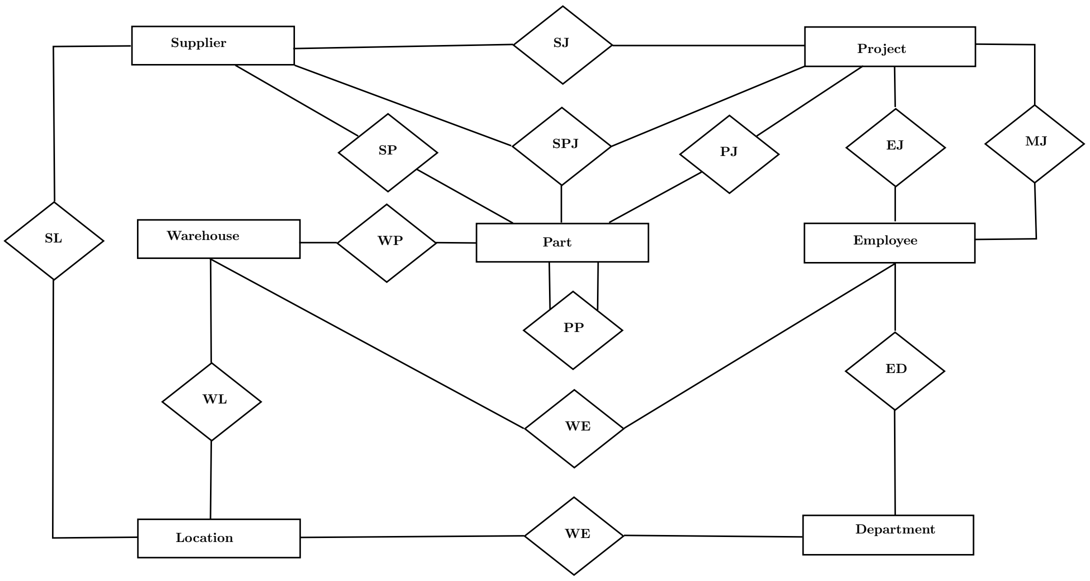


Figure 7: KnowWare Inc. Database

Table 1: Set of five suppliers

Sno	Sname	Status	City
1	Smith	20	London
2	Jones	10	Paris
3	Blake	30	Paris
4	Clark	20	London
5	Adams	30	NULL

Example 1

Considering the relation scheme **"Supplier"** of KnowWare Inc. company database described in Section 2.1. We can write $Supplier = \{Sno, Sname, Status, City\}$ or $Supplier(Sno, Sname, Sta$

The domain of each attribute names might be:

1. $dom(Sno) = dom(status) =$ a set of decimal numbers.
2. $dom(Sname) = dom(City) =$ a set of characters. The **NULL** value occured in the **City** column of Table 1 is discussed in Section 2.2.3.

Table 1 shows a relation of five tuples (or rows). Each row presents one supplier. The first two tuples may be written as $t1 =< 1, Smith, 20, London >$ and $t2 =< 2, Jones, 10, Paris >$.

Table 2: FLIGHTS - Airline schedule.

NUMBER	FROM	TO	DEPARTS	ARRIVES
83	JFK	O'Hare	11:30a	1:43p
84	O'Hare	JFK	3:00p	5:55p
109	JFK	Los Angeles	9:50p	2:52a
213	JFK	Boston	11:43a	12:45p
214	Boston	JFK	2:20p	3:12p

Example 2

Every flight listed in the airline schedule presented in Table 2 has an origin and a destination and it is scheduled to depart at a specific time and arrive at a later time. It has a flight number. The FROM column contains names of airports served by the airline, the ARRIVES column times of day. The order of the columns is immaterial as far as information content is concerned. The DEPARTS and ARRIVES columns could be interchanged with no change in meaning. Finally, since each flight has a unique number, no flight is represented by more than one row.

In the Table 2 the relation scheme is
$FLIGHTS = \{NUMBER, FROM, TO, DEPARTS, ARRIVES\}$.
The domain of each attribute names might be:

1. $dom(NUMBER)$ = a set of decimal numbers.
2. $dom(FROM) = dom(TO) = \{JFK, O'Hare, Boston, LosAngeles, Atlanta\}$.
3. $dom(DEPARTS) = dom(ARRIVES)$ = the set of times of day.

The relation in Table 2 has five tuples. One of them is t_1 defined as $t(NUMBER) = 84$, $t(FROM) = O'Hare$, $t(TO) = JFK$, $t(DEPARTS) = 3:00p$, $t(ARRIVES) = 5:55p$. We can also write $t_1 =< 84, O'Hare, JFK, 3:00p, 5:55p >$.

2.2.2 *Keys*

By definition, the set of all attributes of relation scheme R has the **uniqueness** property, meaning that, at any given time, no two tuples in a relation $r(R)$ at that time are duplicates of one another. Based on this property the following definitions are given.

(a) **Candidate Key** of a relation $r(R)$ is a subset $K = \{B_1, B_2, ..., B_n\}$ of R with the following properties. (i) **uniqueness** which means that for any two distinct tuples t_1 and t_2 in r, $t_1(K) \neq t_2(K)$; (ii) **irreducibility** that is to say no proper subset K' of K has the uniqueness property.

(b) **Super Key** is superset of a candidate key, meaning if r has a candidate key K, and $K \subseteq K'$, then K' is called a *superkey*.

(c) **Primary Key –** A given relation $r(R)$ may have more than one candidate key. The *primary key* is exactly one chosen key of those key, and the others are then called *alternative* keys.

Example

In the FLIGHTS relation shown in Table 2, $\{NUMBER\}$ is a key (and a superkey), so $\{NUMBER, FROM\}$ is a superkey but not a key. And $\{Sno\}$ is a key of the SUPPLIER relation shown in Table 1.

2.2.3 *NULL values in the Tuples*

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases [9].

2.2.4 *Relational Model Notation*

- A relation schema R of degree n is denoted by $R(A_1, A_2, ..., A_n)$.
- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- A relation r on R is written $r(R)$ or $r(A_1A_2...A_n)$.
- To denote the key of a relation, we underline the attribute names in the key. The relation r on scheme $ABCD$ with AC as a key is written $r(\underline{A}B\underline{C}D)$. We can also incorporate the key into the relation scheme R as $R(\underline{A}B\underline{C}D)$. Any relation $r(R)$ is restricted to have
- The letters t, u, v denote tuples.
- An attribute A can be qualified with the relation scheme name R to which it belongs by using the dot notation $R.A$. For example, FLIGHTS.NUMBER.

2.3 INTEGRITY CONSTRAINTS

An **integrity constraint** is a boolean expression that is associated with some database and is required to evaluate at all time to TRUE [4]. A DBMS should provide capabilities for defining and enforcing these constraints [9]. Integrity constraint of the relational model is the part that has changed (and evolved) the most over years. The original emphasis was on **primary** and **foreign keys** specifically(*"keys"* for short). Gradually, however, the importance –indeed, crucial importance– of integrity constraints in general began to be better understood and more widely appreciate [4].

Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry. For typical large applications, it is customary to call such constraints **business rules** [9].

Here are some integrity constraints, expressed in natural language, all based on the KnowWare Inc. database.

1. Every supplier status value is in the range 1 to 100.
2. Every supplier in London has status 20.
3. No two distinct suppliers have the same number.
4. No supplier with status less then 20 supplies any part in a quantity greater than 500.

Integrity constraints can generally be divided into two main categories:

1. Constraints that can be expressed in the database scheme using the DDL (Data Definition Language). These constraints must be formally declared to the DBMS, and the DBMS must then enforce them. For example, the first and last two constraints (number 1, 3 and 4) expressed above belong to this category of constraints.
2. Constraints that cannot be directly expressed in the database scheme. These types of constraints are not understood by the DBMS but they specify what the data means to the **users**. Constraint number 2 belong to this category of constrains.

2.3.1 *Domain constraints*

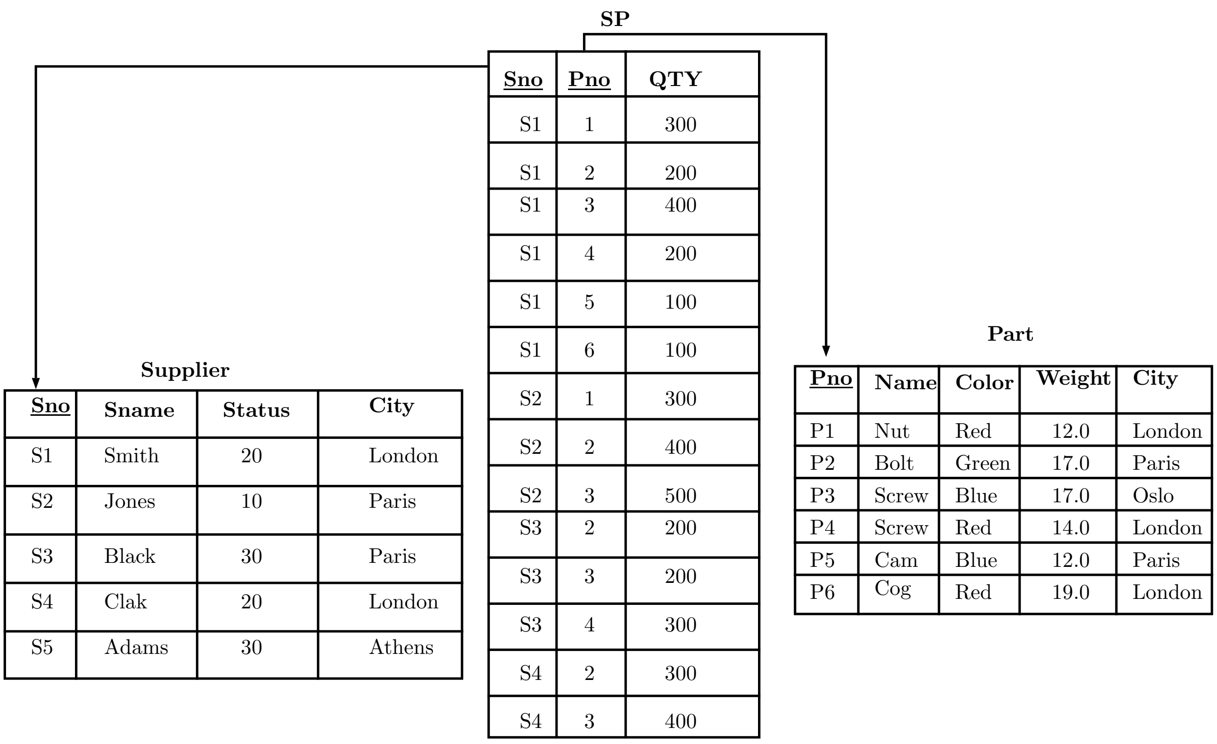
The simplest type of integrity constraint involves specifying a data type for each data item. This kind of constraint is known as **domain constraint**. Domain constraints specify that within each tuple $t =< A_1, A_2, ..., A_n >$, the value of each attribute $A_i, 1 \leqslant i \leqslant n$ must be an atomic value from the domain $dom(A_i)$.

2.3.2 *Key and Primary Key Constraints*

A superkey SK defined in Section 2.2.2 specifies a *uniqueness constraint* that no two distinct tuples t_1 and t_2 in any relation r of relation scheme R can have the same value for SK, $t_1(SK) \neq t_2(SK)$. If this SK is a *key*, it specifies particular uniqueness constraint called ***primary key constraint***.

2.3.3 *Referencial and Foreign Key Constraints*

A more complex frequently type of constraint is the *referential integrity*. A referential constraint involves specifying that each record in a file must be related to one record in other file. A ***foreign key*** is a referencial key that allows to join two tables together by using a *primary key* in one table with a non key field in another table.



SP

<u>Sno</u>	<u>Pno</u>	QTY
S1	1	300
S1	2	200
S1	3	400
S1	4	200
S1	5	100
S1	6	100
S2	1	300
S2	2	400
S2	3	500
S3	2	200
S3	3	200
S3	4	300
S4	2	300
S4	3	400

Supplier

<u>Sno</u>	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Black	30	Paris
S4	Clak	20	London
S5	Adams	30	Athens

Part

<u>Pno</u>	Name	Color	Weight	City
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Figure 8: Supplier, Part and SP relations

Example

Figure 8 shows three relations “Supplier”, “Part”, and “SP”. The “SP” relation contains the quantity of parts should be supplied by each supplier. For example, the first tuple of “SP” shows that the supplier “Smith” (Sno=S1) is under contract to supply 300 parts of type “Nut” (Pno=P1).

Here are three relation schemes Supplier, Part, and SP. The **underlined** attributes constitute in each relation scheme the primary key while the attributes preceded by the symbol # represent the foreign keys. “Sno” and “Pno” attributes constitute together the primary key in “SP” relation scheme. These two attributes are used also to join respectively “Supplier“ and “Part“ relations. For this purpose, two referencial integrity constraints are specified to state that each tuple in ”SPA“ relation must refer to two existing tuples in the ”Supplier“ and ”Part“ relations.

- Supplier(<u>Sno</u>, Sname, Status, City).
- Part(<u>Pno</u>, Name, Color, Weight, City).
- SP(<u>#Sno</u>, <u>#Pno</u>, QTY).

2.3.4 *Constraints on NULL Values*

This constraint specifies whether NULL values are or are not permitted for the tuple value of a particular attribute. For example, if every FLIGHT tuple must have valid, non-

Table 3: Sample values for relation variable SCP

SNO#	CITY	PNO#	QTY
S1	London	P1	90
S1	London	P2	100
S2	Paris	P1	200
S2	Paris	P2	200
S3	Paris	P2	300
S4	London	P2	400
S4	London	P4	400
S4	London	P5	400

NULL values for the FROM and TO attributes, then FROM and TO are constrained to be **NOT NULL**.

2.4 RELATIONAL DATABASES AND DATABASE SCHEME

A **relational database scheme** S is a set of relation schemes $S = \{R_1, R_2, ..., R_m\}$ and a set of **integrity constraints** IC. A relational database state DB of S is a set of relation states $DB = \{r_1, r_2, ..., r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the specified integrity constraints [9].

Example

The set of relational schemes of KnowWare Inc. database described in the Entity/Association Model, presented in Section 2.1, and all integrity constraints may be expressed within it constitute together KnowWare Inc. relational database scheme.

2.5 FUNCTIONAL DEPENDENCIES

Definition : Let X and Y be subsets of the relation scheme R; then the functional dependency (FD) $\mathbf{X} \rightarrow \mathbf{Y}$ holds in R if and only if, whenever two tuples of R agree on X, they also agree on Y. X and Y are the **determinant** and the **dependant**, respectively, and the FD overall can be read as either "**X** functionally determines **Y**" or "**Y** is functionally dependent on **X**", or more simply just "**X** arrow **Y**" [5]

Example

For example, the relation shown in Table 3 satisfies the FD $\{SNO\#\} \rightarrow \{CITY\}$.

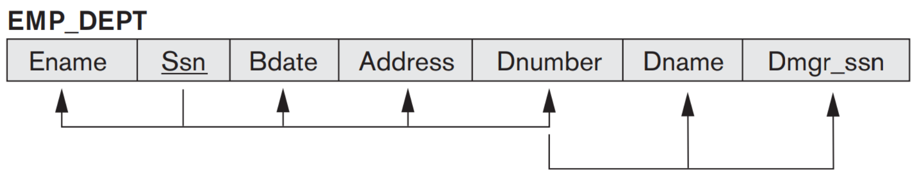


Figure 9: Relation scheme EMP-DEPT

2.5.1 *Closure of a set of functional dependencies*

Definition. Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the **closure** of F. It is denoted by F^+ [9].

To compute F^+ from F, Armstrong [1] gave a set of **inference rules** (more usually called **Armstrong's axioms**) by which new FDs can be inferred from given ones [4].

Let A, B, and C be arbitrary subsets of the set of attributes of the given scheme R, and let us agree to write AB to mean the union of A and B. Then:

1. **Reflexivity:** if B is a subset of A, then $A \rightarrow B$.
2. **Augmentation:** if $A \rightarrow B$, then $AC \rightarrow B$.
3. **Transitivity:** if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Several further rules can be derived from the three given above, the following among them. (Let D is another arbitrary subset of the set of attributes of R)

1. **Self-determination:** $A \rightarrow A$.
2. **Decomposition:** if $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$.
3. **Union:** if $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$.
4. **Composition:** if $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$.

Example

The following set of FDs F is specified on the relation scheme "EMP-DEPT" in Figure 9 [9].

$F = \{Ssn \rightarrow \{Ename, Bdate, Address, Dnumber\}, Dnumber \rightarrow \{Dname, Dmgr\_ssn\}\}$
Some of the additional functional dependencies that we can infer from F are the following:
$Ssn \rightarrow \{Dname, Dmgr\_ssn\}$
$Ssn \rightarrow Ssn$
$Dnumber \rightarrow Dname$

Exercise2.1

Suppose we are given shceme $R(ABCDEF)$ and the FDs:
$\mathbf{A} \rightarrow \mathbf{BC}$

B $\rightarrow$ **E**
CD $\rightarrow$ **EF**
Prove that the FD **AD** $\rightarrow$ **F** $\in$**F⁺** (meaning that we can infer **AD** $\rightarrow$ **F** from **F**).

Answer of exercise 2.1

1. $A \rightarrow BC$ (given)
2. $A \rightarrow C$ (decomposition)
3. $AD \rightarrow CD$ (2, augmentation)
4. $CD \rightarrow EF$ (given)
5. $AD \rightarrow EF$ (3 and 4, transitivity)
6. $AD \rightarrow F$ (5, decomposition)

2.5.2 *Closure of a set of attributes X under a set of FDs F*

Let F a set of FDs over scheme R. If an FD $X \rightarrow Y$ can be *implied* by F, we write $F \models X \rightarrow Y$. To determine if $F \models X \rightarrow Y$, we need only test if $X \rightarrow Y \in F^+$ [12]. Because F^+ can be considerably large than F, we would like to find a means to test if $X \rightarrow Y \in F^+$. The solution is to compute the **closure** X^+ of the set of attributes X under F and test if Y is in X^+. Formally, we write $X \rightarrow Y \in F^+$ if and only if $Y \subseteq X^+$.

The function **CLOSURE(X,F)** given in algorithm 1 returns the closure X^+ under F. The function **MEMBER(F,X**$\rightarrow$ **Y)** given in algorithm 2 uses **CLOSURE** to test the membership of $X \rightarrow Y$ in F^+.

Example

Let $F = \{A \rightarrow D, AB \rightarrow E, BI \rightarrow E, CD \rightarrow I, E \rightarrow C\}$. $\mathrm{CLOSUE}(F, \{A, E\}) = \{A, E\}^+ = \{A, C, D, E, I\}$.

2.5.3 *Cover and equivalence*

Let F and G two sets of FDs over scheme R. If every FD $X \rightarrow Y \in G$ is *implied* by F, we say that G is implied by F and we write $F \models G$.

Definition of cover. Let F and G two sets of FDs. If every FD implied by F is implied by G –i.e., if F^+ is a subset of G^+– we say that G is a *cover* for F [4].

Definition of equivalence. Two sets of FDs F and G over scheme R are *equivalent*, written $F \equiv G$, if and only if $F^+ = G^+$. If $F \equiv G$, then F is a cover for G [12].

Lemma Given sets of FDs over scheme R, $F \equiv G$ if and only if $F \models G$ and $G \models F$ [12].

Algorithm 1 Compute the closure X^+ under the set of FDs F

INPUT: A set of attributes X and a set of FDs F
OUTPUT: X^+ the closure of X under F

```
function CLOSURE(X,F)
    $X^+ \leftarrow X$;
    $\mathtt{old}X^+ \leftarrow \emptyset$;
    while ($\mathtt{old}X^+ \neq X^+$) do
        $\mathtt{old}X^+ \leftarrow X^+$;
        for every FD $Y \rightarrow Z$ in $F$ do
            if $X^+ \supseteq Y$ then
                $X^+ \leftarrow X^+ \cup Z$;
            end if
        end for
    end while
    return $X^+$;
end function
```

Algorithm 2 Test if $F \models X \rightarrow Y$

INPUT: A set of FDs F and an FD $X \rightarrow Y$.
OUTPUT: *true* if $F \models X \rightarrow Y$, *false* otherwise.

```
function MEMBER(F,X → Y)
    if $Y \subseteq \mathrm{CLOSURE}(X, F)$ then
        return true;
    else
        return false;
    end if
end function
```

Exercise2.2

F and G two sets of FDs, $F = \{A \rightarrow BC, A \rightarrow D, CD \rightarrow E\}$ and $G = \{A \rightarrow BCE, A \rightarrow ABD, CD \rightarrow E\}$

Are F and G equivalent?

Exercise2.3

Let F and G two sets of FDs,

- $F = \{A \rightarrow BC, E \rightarrow CF, B \rightarrow E, CD \rightarrow EF\}$.
- $G = \{A \rightarrow B, C \rightarrow DE, AC \rightarrow F\}$.

– Compute the closure $\{A, B\}^+$ under F and $\{A, C\}^+$ under G.

2.5.4 *Irreducible sets of dependencies (or minimal cover)*

Definition A set F of FDs is **irreducible**, called also **minimal cover**, if only if it satisfies the following three properties:

1. The right-hand side (the **dependent**) of every FD in F involves just one attribute (i.e., it is a singleton set).
2. The left-hand side (the **determinant**) of every FD in F is irreducible in turn, meaning that no attribute can be discarded from the determinant without changing the closure F^+ (i.e., without converting F into some set not **equivalent** to F.
3. No FD in F can be discarded from F without changing the closure F^+ (i.e., without converting F into some set not equivalent to F).

Algorithm 3 finds the **minimal cover** F of the set of FDs E.

Exercise2.4

Compute the minimal cover of the set of FDs E on the scheme $R(ABCD)$. $E = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$.

Answer of exercise 2.4

1. Step 1,
   a) $A \rightarrow C$
   b) $A \rightarrow B$ (removed because it occurs twice)
   c) $B \rightarrow C$
   d) $A \rightarrow B$
   e) $AB \rightarrow C$
   f) $AC \rightarrow D$

Algorithm 3 Find the minimal cover F of the set of FDs E

INPUT: A set of FDs E.
OUTPUT: The minimal cover F of E.

```
1:  $F \leftarrow E$;
2:  Replace each FD $X \rightarrow \{A_1, A_2, ..., A_n\}$ in $F$ by the $n$ FDs $X \rightarrow A_1$, $X \rightarrow A_2$, $X \rightarrow A_n$;
3:  for each FD $X \rightarrow A$ in $F$ do
4:      for each attribute $B$ in $X$ do
5:          if $\{\{F - \{X \rightarrow A\}\} \cup \{(X - \{B\}) \rightarrow A\}\}$ is equivalent to $F$ then
6:              replace $X \rightarrow A$ with $(X - \{B\})$ in $F$;
7:          end if
8:      end for
9:  end for
10: for each remaining FD $X \rightarrow A$ in $F$ do
11:     if $\{\{F - \{X \rightarrow A\}\}$ is equivalent to $F$ then
12:         remove $X \rightarrow A$ from $F$;
13:     end if
14: end for
```

2. Step 2,

   $AC \rightarrow D$ (**f**) is replaced by $A \rightarrow D$ because:

   - $A \rightarrow AC$ is implied by composition of $A \rightarrow A$ and $A \rightarrow C$ (**a**).
   - Then, $A \rightarrow D$ is implied by transitivity of $A \rightarrow AC$ and $AC \rightarrow D$ (**f**).

3. Step 3,

   $AB \rightarrow C$ (**e**) is removed because

   - We can imply $AB \rightarrow CB$ by composition of $A \rightarrow C$ (given in **a**) and $B \rightarrow B$.
   - Then, $AB \rightarrow C$ is implied by decomposition of $AB \rightarrow CB$.

4. Step 4,

   $A \rightarrow C$ is removed because it is implied by transitivity of $A \rightarrow B$ (**d**) and $B \rightarrow C$ (**c**)

The minimal cover (or irreducible set) of F is $\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$.

Exercise2.5

Let $F =$
$ABD \rightarrow E$
$AB \rightarrow G$
$B \rightarrow F$
$C \rightarrow J$
$CJ \rightarrow I$
$G \rightarrow H$
Is F an irreducible set ?

NAME	SEXE
{John, Jean, Ivan}	Male
{Mary, Marie}	Female

Table 4: *gender* relation not in 1FN

NAME	SEXE
John	Male
Jean	Male
Ivan	Male
Mary	Female
Marie	Femal

Table 5: *gender* relation in 1FN

2.6 NORMALIZATION

A normal form is a restriction on the database scheme that presumably precludes certain undesirable properties from the database [12].

2.6.1 *Trivial and nontrivial dependencies*

- **Definition:** The FD $X \rightarrow Y$ is trivial if and only if the right-hand Y side is a subset (not necessarily a proper subset) of the left-hand side X. The nontrivial FDs are simply the ones that are not trivial [4].

2.6.2 *First Normal Form (1NF)*

Let relation scheme $R(A_1 A_2 ... A_n)$. A relation $r(R)$ is in *first normal form* (lNF) if and only if for all tuple t appearing in r, the value of each attribute A_i of type $dom(A_i)$ is atomic. To say it in different words, 1NF means that every tuple contains exactly one value for each attribute [4].

Example

The relation *gender*, shown in Table 4, is not in 1NF because it contains values that are sets of atomic values. To be in 1NF, *gender* should be stored like it's in Table 5

2.6.3 *Second Normal Form (2NF)*

- **Definition:** The FD $X \rightarrow Y$ is **irreducible with respect to the scheme R** (or just *irreducible*, if R is understood) if and only if it holds in R and $X^- \rightarrow Y$ doesn't hold in R for any proper subset X^- of X [5].
- **Definition :** A scheme R is in **second normal form** (2NF) if and only if, for every key K of R and every nonkey attribute A of R, the FD $K \rightarrow \{A\}$ (which holds in R, necessarily) is irreducible [5].

SNO#	PNO#	QTY
S1	P1	90
S1	P2	100
S2	P1	200
S2	P2	200
S3	P2	300
S4	P2	400
S4	P4	400
S4	P5	400

Table 6: Sample value for the scheme SC

SNO#	CITY
S1	London
S2	Paris
S3	Paris
S4	London

Table 7: Sample value for the scheme SC

Example

The relation shown in Table 3 contains some values on the scheme $\text{SCP}(\underline{\text{SNO\#}}, \text{CITY}, \underline{\text{PNO\#}}, \text{QTY})$. This scheme clearly suffers from redundancy. Every tuple for supplier $S1$ tells us $S1$ is in London, every supplier for $S2$ tells us $S2$ is in Paris, and so one. The scheme SCP isn't in 2NF because its key is $\{SNO, PNO\}$ and the FD $\{SNO, PNO\} \to CITY$ isn't irreducible. We can drop PNO from the determinant and what remains, $\{SNO\} \to \{CITY\}$, is still and FD that holds in SCP.

To be in 2NF, SCP should be divided into two new relation schemes $\text{SPQ}(\underline{\text{SNO\#}}, \underline{\text{PNO\#}}, \text{QTY})$ and $\text{SC}(\underline{\text{SNO\#}}, \text{CITY})$. Table 6 and Table 7 show sample values for new schemes SPQ and SC.

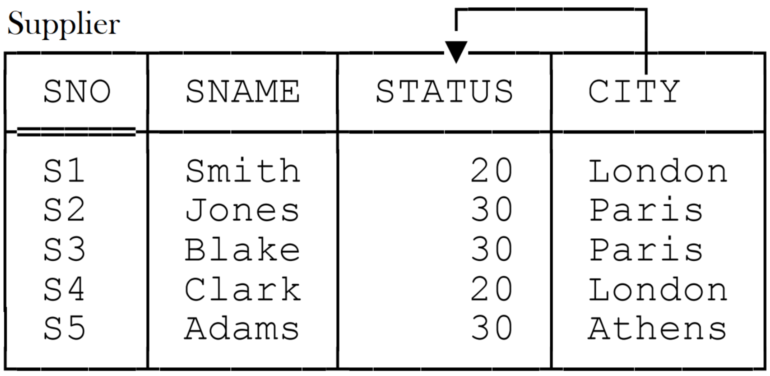
2.6.4 *Third Normal Form (3NF)*

- **Definition:** The scheme R is in third normal form (3NF) if and only if, for every nontrivial FD $X \to Y$ that holds in R, either (a) X is a superkey or (b) Y is a subkey [5].
- Contrary to definitions of 3NF commonly found in the literature that take often the form "R is in 3NF if and only if it's in 2NF and ...", Date C. J. [5] prefer a definition that takes non mention of 2NF. This definition also follows that 3NF implies 2NF.

Example

The scheme Supplier in Figure 10 isn't in 3NF. It's clear that the FD $CITY \to STATUS$ holds in that scheme. $CITY$ isn't a superkey and $STATUS$ isn't a subkey.

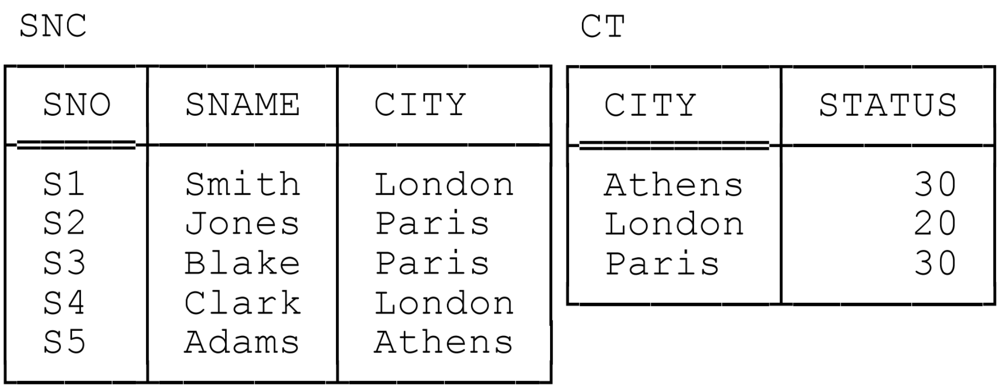
To be in 3NF, this scheme will be subdivided into two new schemes like they are shown in Figure 11.



Supplier

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	30	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Figure 10: The supplier scheme-sample value



SNC

SNO	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S3	Blake	Paris
S4	Clark	London
S5	Adams	Athens

CT

CITY	STATUS
Athens	30
London	20
Paris	30

Figure 11: The supplier scheme-sample value

SNP	SNO	SNAME	PNO	QTY
	S1	Smith	P1	300
	S1	Smith	P2	200
	S1	Smith	P3	400
	..	.....	..	...
	S2	Jones	P1	300
	S2	Jones	P2	400
	..	.....	..	...

Figure 12: The SNP scheme-sample value

2.6.5 *Boyce/Codd Normal Form (BCNF)*

- **Definition:** A scheme R is in **Boyce/Codd Normal Form** (BCNF) if and only if, for every nontrivial FD $X \rightarrow Y$ that holds in R, X is a superkey [5].
- This definition takes no mention of 2NF and 3NF. Note, however, that it can be derived from the definition of 3NF by dropping of (b) condition ("Y is a subkey"). But it's clear it follows that if a scheme R is in BCNF, then it's certainly in 3NF.

Example

The scheme $SNP(SNO, SNAME, PNOQTY)$ has two candidate keys $\{SNO, PNO\}$ and $\{SNAME, PNO\}$; for that reason there's no underline attributes. The FD $\{SNO\} \rightarrow \{SNAME\}$ and $\{SNAME\} \rightarrow \{SNO\}$ hold in SNP. In both cases the determinant isn't a superkey, and so the scheme SNP isn't in BCNF.

To be in BCNF we decompose SNP into either of the following decompositions.

- **Decomposition 1.** $\{SNO, SNAME\}$ and $\{SNO, PNO, QTY\}$.
- **Decomposition 2.** $\{SNO, SNAME\}$ and $\{SNAME, PNO, QTY\}$.

2.7 SQL LANGUAGE

SQL (Structured Query Language) is a standard programming language used for managing and manipulating data in relational databases like MySQL, Oracle, SQL Server, PostGreSQL. It is used to create, modify, and query databases and is widely used in modern data-driven applications. SQL is used to communicate with databases, allowing users to retrieve, insert, update, and delete data. SQL is also used for database management tasks such as creating tables, defining relationships between tables, and setting constraints on data. SQL is a powerful and versatile language, and its popularity has made it a critical skill for anyone working with data or databases.

SQL, which stands for Structured Query Language, was first developed in the early 1970s by Donald D. Chamberlin and Raymond F. Boyce at IBM. The initial version of SQL, called SEQUEL (Structured English Query Language), was designed as a way to access and manipulate data in IBM's experimental System R database.

In the late 1970s and early 1980s, SQL was standardized by ANSI (American National Standards Institute) and ISO (International Organization for Standardization), which helped to establish it as the dominant language for managing relational databases. The first ANSI SQL standard was published in 1986, followed by subsequent revisions and updates over the years.

As relational databases became more popular in the 1980s and 1990s, SQL evolved to support a wider range of functionality, including data manipulation, data definition, and data control. In addition, various commercial database management systems (DBMS) were developed that supported SQL, such as Oracle, IBM DB2, and Microsoft SQL Server.

2.7.1 *SQL basics*

Syntax of SQL statements The syntax of SQL statements refers to the specific rules and conventions used to write SQL commands and queries that interact with a database. Proper syntax is essential for ensuring that SQL commands are executed correctly and that the resulting data is accurate and useful. Some key elements of SQL syntax include:

1. Clauses: SQL commands typically consist of one or more clauses that specify the action to be taken. For example, the SELECT command consists of the SELECT and FROM clauses, which specify which columns to retrieve and from which table.

2. Keywords: SQL commands are made up of keywords that define the action to be taken, such as SELECT, INSERT, UPDATE, DELETE, and JOIN.

3. Operators: SQL commands use various operators, such as comparison operators (=, <>, >, <, etc.) and logical operators (AND, OR, NOT), to filter and manipulate data.

4. Values and expressions: SQL statements often require the use of values and expressions, such as dates, strings, and mathematical expressions, to perform calculations or filter data.

5. Parentheses: Parentheses are used to group clauses, expressions, and operators in SQL statements to ensure that the order of operations is correct.

Overall, understanding the syntax of SQL statements is critical for writing correct and efficient SQL code. With proper syntax, SQL statements can be used to retrieve, manipulate, and analyze data from relational databases.

2.7.1.1 *Data types in SQL*

Data types in SQL refer to the categories of data that can be stored and manipulated in a relational database. SQL supports various data types that are used to define the structure and behavior of data in a table. Some of the commonly used data types in SQL include:

1. Numeric data types: Numeric data types are used to store numeric values such as integers, decimals, and floating-point numbers. Examples of numeric data types in SQL include INT, BIGINT, DECIMAL, and FLOAT.

2. Character data types: Character data types are used to store text values such as names, addresses, and descriptions. Examples of character data types in SQL include VARCHAR, CHAR, and TEXT.

3. Date and time data types: Date and time data types are used to store dates and times. Examples of date and time data types in SQL include DATE, DATETIME, and TIMESTAMP.

4. Boolean data types: Boolean data types are used to store true/false values. In SQL, the BOOLEAN data type is represented by BIT or BOOL.

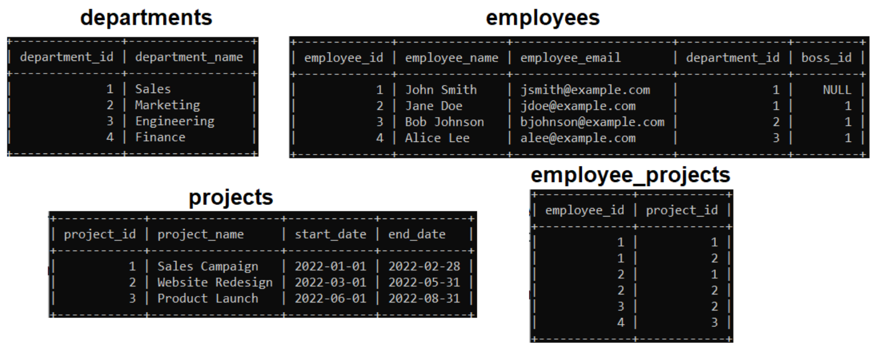
5. Binary data types: Binary data types are used to store binary data such as images or files. Examples of binary data types in SQL include BLOB and VARBINARY.

6. Other data types: SQL also supports other data types such as XML, JSON, and ARRAY.

Understanding data types in SQL is important for designing efficient and accurate database schemas. By choosing the appropriate data type for each column in a table, you can ensure that the data is stored and manipulated correctly and that the database can perform efficiently.

2.7.2 *Data Retrieval with SQL*

We use in this section the database demonstrated in Figure 13. These database describes a set of employees that work at a set of departments and assigned to different projects.



```
+---------------+-----------------+
| department_id | department_name |
+---------------+-----------------+
|             1 | Sales           |
|             2 | Marketing       |
|             3 | Engineering     |
|             4 | Finance         |
+---------------+-----------------+
```

```
+-------------+---------------+----------------------+---------------+---------+
| employee_id | employee_name | employee_email       | department_id | boss_id |
+-------------+---------------+----------------------+---------------+---------+
|           1 | John Smith    | jsmith@example.com   |             1 |    NULL |
|           2 | Jane Doe      | jdoe@example.com     |             1 |       1 |
|           3 | Bob Johnson   | bjohnson@example.com |             2 |       1 |
|           4 | Alice Lee     | alee@example.com     |             3 |       1 |
+-------------+---------------+----------------------+---------------+---------+
```

```
+------------+-----------------+------------+------------+
| project_id | project_name    | start_date | end_date   |
+------------+-----------------+------------+------------+
|          1 | Sales Campaign  | 2022-01-01 | 2022-02-28 |
|          2 | Website Redesign| 2022-03-01 | 2022-05-31 |
|          3 | Product Launch  | 2022-06-01 | 2022-08-31 |
+------------+-----------------+------------+------------+
```

```
+-------------+------------+
| employee_id | project_id |
+-------------+------------+
|           1 |          1 |
|           1 |          2 |
|           2 |          1 |
|           2 |          2 |
|           3 |          2 |
|           4 |          3 |
+-------------+------------+
```

Figure 13: Employees database

SELECT statements

1. Retrieve all columns from the "employees" table:

```
SELECT * FROM employees;
```

2. Retrieve the names and email addresses of all employees:

   ```
   SELECT employee_name, employee_email FROM employees;
   ```

WHERE clause

1. Retrieve the names and email addresses of all employees who are not working on any projects:

   ```
   SELECT employee_name, employee_email
   FROM employees
   WHERE employee_id NOT IN (SELECT employee_id FROM employee_projects);
   ```

2. Retrieve the names and dates of all projects that are currently active:

   ```
   SELECT project_name, start_date, end_date
   FROM projects
   WHERE end_date >= CURRENT_DATE();
   ```

Joins in SQL

SQL provides several types of join operations to combine data from two or more tables. Some of the most commonly used join operations in SQL are:

1. Inner Join: Returns only the rows that have matching values in both tables being joined.
2. Left Join: Returns all the rows from the left table and the matching rows from the right table. If there are no matching rows in the right table, it returns NULL values.
3. Right Join: Returns all the rows from the right table and the matching rows from the left table. If there are no matching rows in the left table, it returns NULL values.
4. Full Outer Join: Returns all the rows from both tables and combines the rows with matching values. If there are no matching rows in one of the tables, it returns NULL values for the missing values.
5. Cross Join: Returns the Cartesian product of the two tables, i.e., all possible combinations of rows from the two tables.
6. Self Join: Joins a table with itself to create a new table that contains only the rows where a certain condition is true.

The first four joins operations are illustrated in Figure 14.

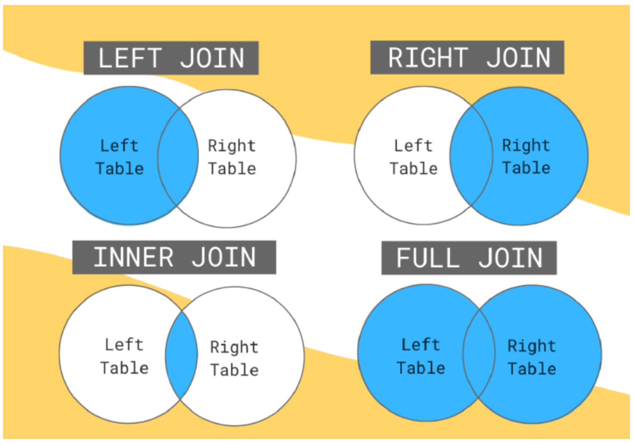


Figure 14: SQL joins

The following query joins the employees table with the departments table based on the department_id column. It selects the employee_name column from the employees table and the department_name column from the departments table where there is a match between the department_id values in both tables.

```
SELECT e.employee_name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.department_id;
```

GROUP BY clause

The following query groups the employees by their department and returns the number of employees in each department. The GROUP BY clause is used to group the data by the department_id column, and the COUNT(∗) function is used to count the number of employees in each group. The output will show each department ID and the corresponding number of employees.

```
SELECT department_id, COUNT(*) as num_employees
FROM employees
GROUP BY department_id;
```

Having clause

This query groups the employees by their department and returns only the departments with more than 3 employees. The GROUPBY clause is used to group the data by the department_id column, and the COUNT(∗) function is used to count the number of employees in each group. The HAVING clause is then used to filter the results to only show departments with more than 3 employees.

Note that the HAVING clause is similar to the WHERE clause, but it is used specifically for filtering the results of grouped data. The HAVING clause is evaluated after the GROUP BY clause, so it can be used to filter the results based on the results of the grouping function.

```
SELECT department_id, COUNT(*) as num_employees
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 3;
```

ORDER BY clause

The ORDER BY clause is typically used in conjunction with the SELECT statement to sort the results based on one or more columns. The default ordering direction is ascending (ASC), but you can specify DESC to sort the results in descending order. You can also specify multiple columns in the ORDER BY clause to sort the results by multiple criteria.

The following query selects the employee_name and salary columns from the employees table and orders the results in descending order by the salary column. The ORDER BY clause is used to sort the results based on the specified column and ordering direction ("DESC" in this case). The output will show the employee names and salaries in descending order, with the highest paid employees first.

```
SELECT employee_name, salary
FROM employees
ORDER BY salary DESC;
```

2.7.3 *Data Manipulation with SQL*

INSERT statement

The following INSERT statement adds three new employees to the employees table in a single query. Each set of values is separated by a comma, and the column names are specified in parentheses before the VALUES keyword. Note that the order of the values in each set must match the order of the columns specified in the INSERT INTO clause.

```
INSERT INTO employees (employee_name, department_id, salary)
VALUES ('Jane Smith', 1, 60000),
       ('Bob Johnson', 3, 45000),
       ('Sara Lee', 2, 55000);
```

DELETE clause

DELETE clause is used to delete one or more rows based on some predicates. The following DELETE statement deletes a single row from the employees table where the employee_id is equal to 5. The DELETE FROM clause specifies the table to delete the row from, and the WHERE clause is used to specify the condition for which rows to delete.

```
DELETE FROM employees
WHERE employee_id = 5;
```

You can also delete multiple rows at once by specifying a more general condition in the WHERE clause. For example, to delete all employees with a salary less than 40,000:

```
DELETE FROM employees
WHERE salary < 40000;
```

2.8 conclusion

In this chapter, we delved into the world of relational databases, which are an essential component of modern data storage and retrieval systems. We discussed the key principles of functional dependencies and normalization, which are critical for ensuring data integrity, minimizing redundancy, and optimizing performance. We explored the normalization process, which involves breaking down large tables into smaller, more specific tables to eliminate repeating groups and reduce the risk of anomalies.

We also covered SQL, which is the primary language used for interacting with relational databases. We discussed how SQL enables users to create and modify tables, insert, update, and delete data, and perform complex queries to extract information from the database. We highlighted some of the key features of SQL, including its ability to handle complex join operations, grouping and aggregation, sorting, and filtering.

Overall, this chapter provides a comprehensive overview of relational databases and their key components, including functional dependencies, normalization, and SQL. By understanding these principles, readers will be better equipped to design and implement robust, scalable, and efficient data storage and retrieval systems.

3 OBJECT AND OBJECT-RELATIONAL DATABASES

Object-Oriented Databases, actually known as **Object Databases** (**ODB**), were originally developed as an alternative to relational database technology for the modeling, storage, and access of complex data forms that were increasingly found in advanced applications of database technology like *Engineering Design*, *Geographical Information Systems*, and *Multimedia*. After much debate regarding Object-Oriented versus relational database technology, object features were eventually incorporated into relational database systems to create a new model and systems characterized as **Object-Relational Databases** (**ORDB**) and **Object Relational Database Management Systems** (**ORDBMS**). Another reason for creation of ODB is the fast increasing of the use of Object-Oriented Programming Languages (**OOPL**) for development of those complex applications, so ODB can be directly used to store and retrieve object data of these kind of applications.

3.1 OBJECT TECHNOLOGY

The term *object-oriented*, abbreviated OO, were originally used in the OO Programming Languages, or OOPLs like Java, C++, C#. Today, OO concepts are incorporated into different areas of software engineering, multi-agent systems, databases and computer systems in general [9]. The main idea of the OO paradigm is to couple the data with behavior. Simula was designed in the late 1960s in *Norway* as the first procedural language to do this task and used, as its name suggests, for doing simulations [2]. Simula language introduced objects, classes, inheritance, subclasses, and many other OO features required for any OOPL.

Once the advantages of the use of object-oriented approach instead of procedural one were recognized, Simula rapidly developed into a general-purpose programming language [11]

The following sections describes the fundamental concepts commonly used in object technology, that are *objects*, *encapsulation*, *classes*, and *inheritance*.

3.1.1 *Objects, Properties, Methods, and Messages*

Object is the most important concept in object paradigm. Object is an encapsulated unit of data (or properties) and behavior (or methods). *Encapsulation* means that the internal implementation is covered and neither program code or variables are visible outside the object. Rather, objects have public interface consisting of public *methods* and *properties*. In general, properties cannot be accessible and manipulated directly but only through data access methods.

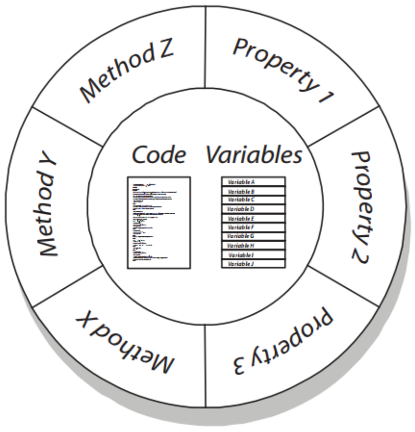


Figure 15: An *object* – codes, variables and methods [11]

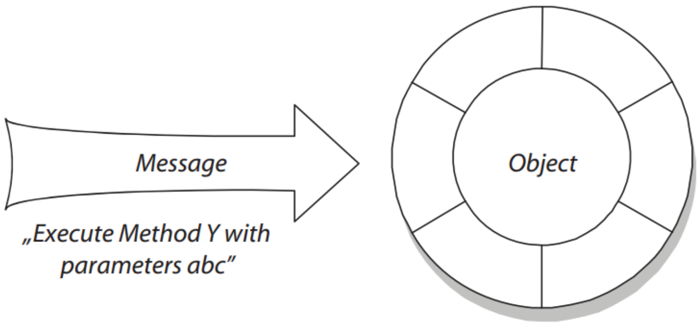


Figure 16: Sending a message to an object [11]

To execute a method, we need to send the object a *message* including the method name and the set of parameters required for the method execution. The object invokes immediately the desired method and passes it parameter values.

3.1.2 *Object Classes and Instances*

An *object class*, or simply *class*, is a blueprint to build a specific type of object. It defines how an object will behave and what the object will contain. The incarnations of classes are called *instances* of these classes. For example, the *circle class* defines the class of geometric *circles*. Each circle is defined by one center point characterized by two coordinates x and y and one radius r. The behavior is guarantee by a set of methods such as *Draw*, *Move*, *Color*..etc. Figure 17 demonstrates the class "Circle" and three different instances.

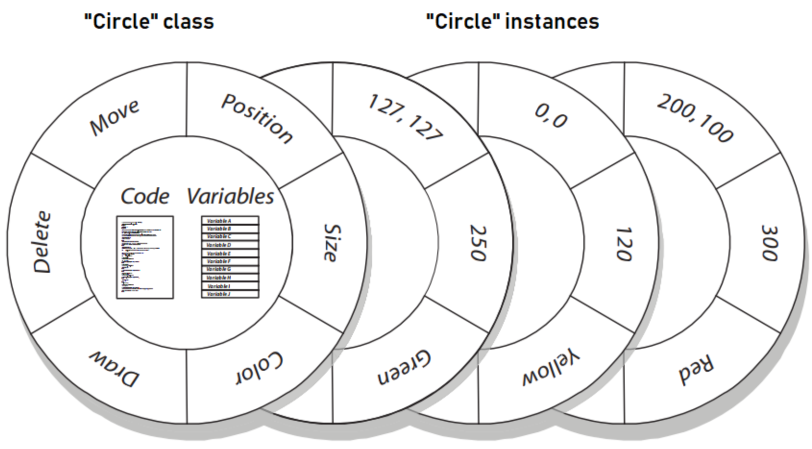


Figure 17: "Circle" class and instances

3.1.3 *Inheritance, Specialization, and Generalization*

Inheritance

Inheritance is the mechanism by which more specific elements incorporate structure and behavior defined by more general elements [15].

Inheritance is one of the most important feature of Object Oriented Programming. It is the capability of derivation of a class B from another class A. The *subclass* B has a "**is a**" relationship to the *super class* A. In this case, we say that class B extends class A, so B inherits from A all its properties and methods. Thus, the subclass has the same interface as the super class and can be used in its place. This principle is called *substitutability* and means that every message you can send to an instance of the super class is also valid for instances of the subclass [11].

Example

Figure 18 demonstrates inheritance relationships between some classes. The class "Form" is the super class of three subclasses "Triangle", "Ellipse", and "Rectangle". All these classes model geometric forms. Using the "**is a**" relationship we can simply say for example – *a "Triangle" is a "Form"*.

Specialization

A subclass B can be specialized by adding new properties and methods that the super class A does not have. Similarly, you can overwrites inherited methods so the subclass behaviors differently than the super class.

Generalization

The generalization relationship is a taxonomic relationship between a more general description and a more specific description that extends it. The more general description

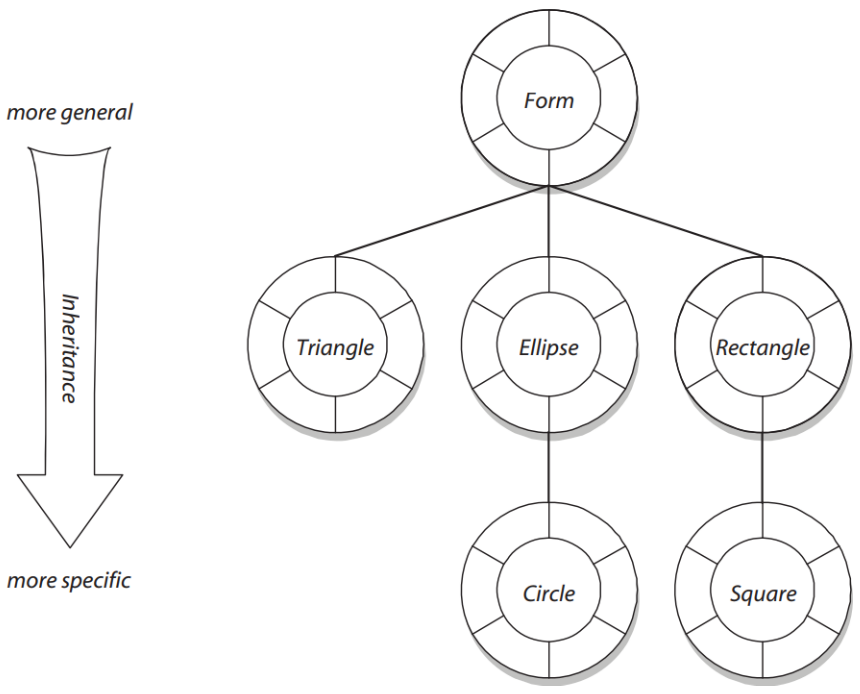


Figure 18: Inheritance, Specialization, and Generalization [11]

is called the *parent*; an element in the transitive closure is an ancestor. The more specific description is called the *child*; an element in the transitive closure is a descendant [**?**]. The super class "Form" appeared in Figure 18 is more general than their children classes "Triangle", "Ellipse", and "Rectangle" and all other inherited classes.

3.1.4 *Object persistence and other ODB features*

3.1.4.1 *Persistence*

From the OOPL point of view, objects can be instantiated in the RAM memory during the execution time of one program meaning that all these objects disappear when the program execution ends.

Persistence denotes that an object can continue to exist after the program that instantiated it ends or even after the system that runs the program turned off.

Not all objects are meant to be stored permanently in the Object Database, there are two types of objects **persistent** and **transient**; the typical mechanisms to make objects persistent are **Naming** and **Reachability**.

3.1.4.2 *Object Naming and Reachability*

NAMING persistent object is specified via specific statement or operation in program; persistent objects are used as entry points into the ODB. However, it is not practical to explicitly give persistent names to all objects in ODB. Thus, a more practical mechanism is **Reachability**

REACHABILITY object is persistent by reachability if a sequence of references in the object graph leads to a persistent (named) object. Reachability involves giving an object a unique persistent name within a particular ODB. This mechanism works by making the object reachable from some other persistent object. An object B is said to be reachable from an object A if a sequence of references in the database lead from object A to object B.

3.1.4.3 *Extent Objects*

Extent denotes a collection of objects of the same type [7].

3.1.5 *Complex Type Structures for Objects*

In ODBs, a complex type may be constructed from other types by nesting of type constructors. The three most basic constructors are atom, struct (or tuple), and collection.

1. The term **atom** is not used in the latest object standard. They are called **single-valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value. are similar to the basic types in many programming languages: integers, strings, floating point numbers, enumerated types, Booleans, and so on.
2. **struct** (or **tuple**) can create standard structured types. such as the tuples (record types) in the basic relational model. It is considered to be a **type generator**. For example, two different structured types that can be created are: struct Name<FirstName: string, MiddleInitial: char, LastName: string>, and struct CollegeDegree<Major: string, Degree: string, Year: date>. To create complex nested type structures in the object model, the **collection** type constructors are needed, which we discuss next.
3. **Collection** (or multivalued) type constructors include the **set(T)**, **list(T)**, **bag(T)**, **array(T)**, and **dictionary(K,T)** type constructors. These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be type generators because many different types can be created. For example, set(string), set(integer), and set(Employee) are three different types that can be created from the set type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set(string) must be string values.

Example

An **object definition language (ODL)** [1] that incorporates the preceding type constructors can be used to define the object types for a particular database application. In Figure 19 the attributes that refer to other objectssuch as Dept of EMPLOYEE or Projects of DEPARTMENTare basically OIDs that serve as references to other objects to represent *relationships* among the objects.

[1]This corresponds to the DDL (data definition language) of the relational database system.

```
define type EMPLOYEE
    tuple (  Fname:        string;
             Minit:        char;
             Lname:        string;
             Ssn:          string;
             Birth_date:   DATE;
             Address:      string;
             Sex:          char;
             Salary:       float;
             Supervisor:   EMPLOYEE;
             Dept:         DEPARTMENT;

define type DATE
    tuple (  Year:         integer;
             Month:        integer;
             Day:          integer;  );

define type DEPARTMENT
    tuple (  Dname:        string;
             Dnumber:      integer;
             Mgr:          tuple (  Manager:      EMPLOYEE;
                                    Start_date:   DATE;  );
             Locations:    set(string);
             Employees:    set(EMPLOYEE);
             Projects:     set(PROJECT);  );
```

Figure 19: Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

3.2 OBJECT-RELATIONAL FEATURES: OBJECT DATABASE EXTENSIONS TO SQL

3.2.1 *SQL/Foundation*

SQL was first specified in (1974) and continued its evolution with a new standard, initially called SQL3 while being developed, and later known as SQL:99 for the parts of SQL3 that were approved into the standard. Starting with the version of SQL known as SQL3, features from object databases were incorporated into the SQL standard. At first, these extensions were known as SQL/Object, but later they were incorporated in the main part of SQL, known as SQL/Foundation. [9].

The following are some of the object database features that have been included in SQL:

- Some type constructors have been added to specify complex objects. These include the row type, which corresponds to the tuple (or struct) constructor. An *array* type for specifying collections is also provided. Other collection type constructors, such as *set*, *list*, and *bag* constructors, were not part of the original SQL/Object specifications but were later included in the standard.
- **Encapsulation of operations** is provided through the mechanism of userdefined types (UDTs) that may include operations as part of their declaration. These are somewhat similar to the concept of abstract data types that were developed in programming languages. In addition, the concept of user defined routines (UDRs) allows the definition of general methods (operations).
- Inheritance mechanisms are provided using the keyword **UNDER**.

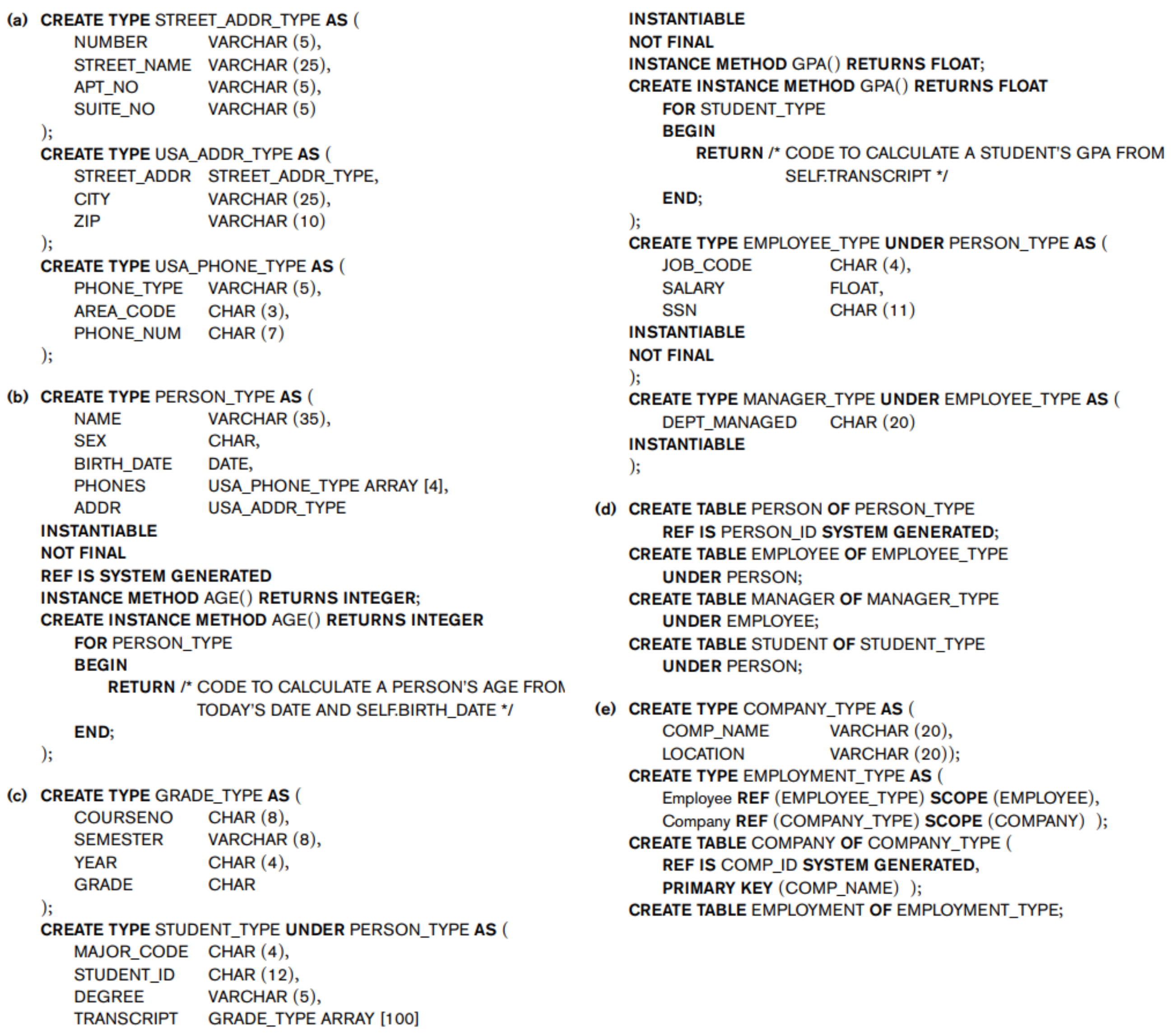
Example

Figure 20 some of the object features of SQL. (a) Using UDTs as types for attributes such as Address and Phone, (b) Specifying UDT for PERSON_TYPE, (c) Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE, Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE, (d) Creating tables based on some of the UDTs,and illustrating table inheritance, (e) Specifying relationships using REF and SCOPE.

by using the keyword **ROW**. For example, we could use the following instead of declaring STREET_ADDR_TYPE as a separate type as in Figure 20.

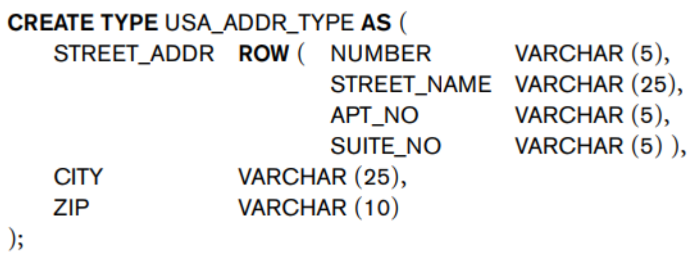
3.2.2 *The Object Definition Language ODL*

The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular programming language. Its main use is to create object specificationsthat is, classes and interfaces. Hence, ODL is not a full programming language. A user can specify a database schema in ODL independently of any programming language, and then use the specific language bindings to specify how ODL constructs



```
(a) CREATE TYPE STREET_ADDR_TYPE AS (
        NUMBER          VARCHAR (5),
        STREET_NAME     VARCHAR (25),
        APT_NO          VARCHAR (5),
        SUITE_NO        VARCHAR (5)
    );
    CREATE TYPE USA_ADDR_TYPE AS (
        STREET_ADDR     STREET_ADDR_TYPE,
        CITY            VARCHAR (25),
        ZIP             VARCHAR (10)
    );
    CREATE TYPE USA_PHONE_TYPE AS (
        PHONE_TYPE      VARCHAR (5),
        AREA_CODE       CHAR (3),
        PHONE_NUM       CHAR (7)
    );

(b) CREATE TYPE PERSON_TYPE AS (
        NAME            VARCHAR (35),
        SEX             CHAR,
        BIRTH_DATE      DATE,
        PHONES          USA_PHONE_TYPE ARRAY [4],
        ADDR            USA_ADDR_TYPE
    INSTANTIABLE
    NOT FINAL
    REF IS SYSTEM GENERATED
    INSTANCE METHOD AGE() RETURNS INTEGER;
    CREATE INSTANCE METHOD AGE() RETURNS INTEGER
        FOR PERSON_TYPE
        BEGIN
            RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                    TODAY'S DATE AND SELF.BIRTH_DATE */
        END;
    );

(c) CREATE TYPE GRADE_TYPE AS (
        COURSENO        CHAR (8),
        SEMESTER        VARCHAR (8),
        YEAR            CHAR (4),
        GRADE           CHAR
    );
    CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
        MAJOR_CODE      CHAR (4),
        STUDENT_ID      CHAR (12),
        DEGREE          VARCHAR (5),
        TRANSCRIPT      GRADE_TYPE ARRAY [100]
    INSTANTIABLE
    NOT FINAL
    INSTANCE METHOD GPA() RETURNS FLOAT;
    CREATE INSTANCE METHOD GPA() RETURNS FLOAT
        FOR STUDENT_TYPE
        BEGIN
            RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                    SELF.TRANSCRIPT */
        END;
    );
    CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
        JOB_CODE        CHAR (4),
        SALARY          FLOAT,
        SSN             CHAR (11)
    INSTANTIABLE
    NOT FINAL
    );
    CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
        DEPT_MANAGED    CHAR (20)
    INSTANTIABLE
    );

(d) CREATE TABLE PERSON OF PERSON_TYPE
        REF IS PERSON_ID SYSTEM GENERATED;
    CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
        UNDER PERSON;
    CREATE TABLE MANAGER OF MANAGER_TYPE
        UNDER EMPLOYEE;
    CREATE TABLE STUDENT OF STUDENT_TYPE
        UNDER PERSON;

(e) CREATE TYPE COMPANY_TYPE AS (
        COMP_NAME       VARCHAR (20),
        LOCATION        VARCHAR (20));
    CREATE TYPE EMPLOYMENT_TYPE AS (
        Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
        Company REF (COMPANY_TYPE) SCOPE (COMPANY) );
    CREATE TABLE COMPANY OF COMPANY_TYPE (
        REF IS COMP_ID SYSTEM GENERATED,
        PRIMARY KEY (COMP_NAME) );
    CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
```

Figure 20: Type and table creation



```
CREATE TYPE USA_ADDR_TYPE AS (
    STREET_ADDR   ROW (  NUMBER        VARCHAR (5),
                         STREET_NAME   VARCHAR (25),
                         APT_NO        VARCHAR (5),
                         SUITE_NO      VARCHAR (5) ),
    CITY          VARCHAR (25),
    ZIP           VARCHAR (10)
);
```

Figure 21: ROW instead of declaring new type

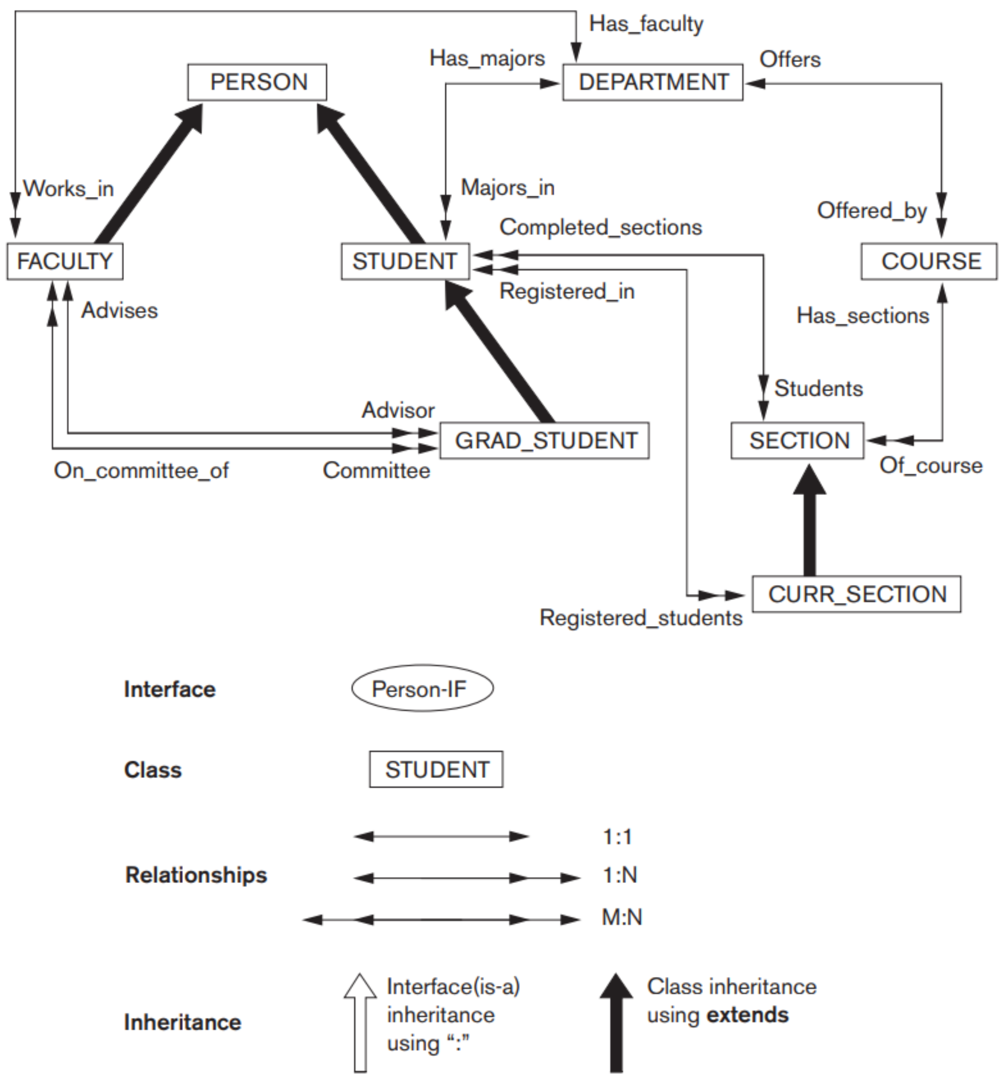


Figure 22: UNIVERSITY database

can be mapped to constructs in specific programming languages, such as C++, Smalltalk, and Java.

Figure 22 shows a possible object schema for part of the UNIVERSITY database.

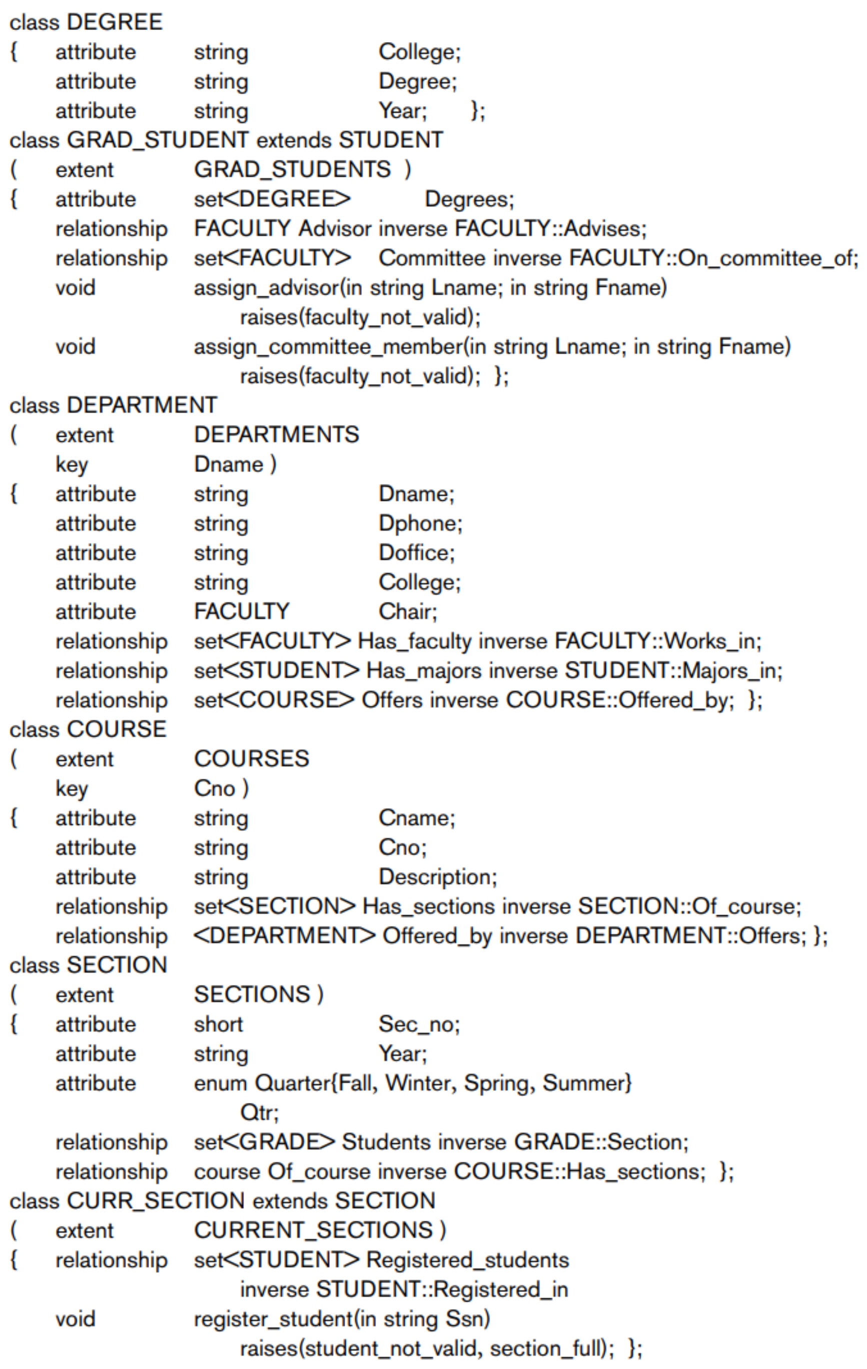
Figure 23 and Figure 24 show the straightforward way of mapping part of the UNIVERSITY database.

3.2.3 *The Object Query Language OQL*

The object query language OQL is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, Smalltalk, and Java.

```
class PERSON
(    extent         PERSONS
     key            Ssn  )
{    attribute      struct Pname {   string    Fname,
                                     string    Mname,
                                     string    Lname  }     Name;
     attribute      string                                  Ssn;
     attribute      date                                    Birth_date;
     attribute      enum Gender{M, F}                       Sex;
     attribute      struct Address {  short    No,
                                      string   Street,
                                      short    Apt_no,
                                      string   City,
                                      string   State,
                                      short    Zip  }       Address;
     short          Age();    };
class FACULTY extends PERSON
(    extent         FACULTY  )
{    attribute      string           Rank;
     attribute      float            Salary;
     attribute      string           Office;
     attribute      string           Phone;
     relationship   DEPARTMENT       Works_in inverse DEPARTMENT::Has faculty;
     relationship   set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
     relationship   set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
     void           give_raise(in float raise);
     void           promote(in string new rank);  };
class GRADE
(    extent         GRADES  )
{
     attribute      enum GradeValues{A,B,C,D,F,I, P} Grade;
     relationship   SECTION Section inverse SECTION::Students;
     relationship STUDENT Student inverse STUDENT::Completed_sections;  };
class STUDENT extends PERSON
(    extent         STUDENTS  )
{    attribute      string           Class;
     attribute      DEPARTMENT          Minors_in;
     relationship   DEPARTMENT Majors_in inverse DEPARTMENT::Has_majors;
     relationship   set<GRADE> Completed_sections inverse GRADE::Student;
     relationship   set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
     void           change_major(in string dname) raises(dname_not_valid);
     float          gpa();
     void           register(in short secno) raises(section_not_valid);
     void           assign_grade(in short secno; IN GradeValue grade)
                        raises(section_not_valid,grade_not_valid);  };
```

Figure 23: Possible ODL schema for the UNIVERSITY database (Part 1)



```
class DEGREE
{   attribute     string              College;
    attribute     string              Degree;
    attribute     string              Year;     };
class GRAD_STUDENT extends STUDENT
(   extent        GRAD_STUDENTS )
{   attribute     set<DEGREE>              Degrees;
    relationship  FACULTY Advisor inverse FACULTY::Advises;
    relationship  set<FACULTY>    Committee inverse FACULTY::On_committee_of;
    void          assign_advisor(in string Lname; in string Fname)
                      raises(faculty_not_valid);
    void          assign_committee_member(in string Lname; in string Fname)
                      raises(faculty_not_valid);  };
class DEPARTMENT
(   extent        DEPARTMENTS
    key           Dname )
{   attribute     string              Dname;
    attribute     string              Dphone;
    attribute     string              Doffice;
    attribute     string              College;
    attribute     FACULTY             Chair;
    relationship  set<FACULTY> Has_faculty inverse FACULTY::Works_in;
    relationship  set<STUDENT> Has_majors inverse STUDENT::Majors_in;
    relationship  set<COURSE> Offers inverse COURSE::Offered_by;  };
class COURSE
(   extent        COURSES
    key           Cno )
{   attribute     string              Cname;
    attribute     string              Cno;
    attribute     string              Description;
    relationship  set<SECTION> Has_sections inverse SECTION::Of_course;
    relationship  <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };
class SECTION
(   extent        SECTIONS )
{   attribute     short               Sec_no;
    attribute     string              Year;
    attribute     enum Quarter{Fall, Winter, Spring, Summer}
                      Qtr;
    relationship  set<GRADE> Students inverse GRADE::Section;
    relationship  course Of_course inverse COURSE::Has_sections;  };
class CURR_SECTION extends SECTION
(   extent        CURRENT_SECTIONS )
{   relationship  set<STUDENT> Registered_students
                      inverse STUDENT::Registered_in
    void          register_student(in string Ssn)
                      raises(student_not_valid, section_full);  };
```

Figure 24: Possible ODL schema for the UNIVERSITY database (Part 2)

The basic OQL syntax is a select ... from ... where ... structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of 'Engineering' can be written as follows:

```
Q0: select  D.Dname
    from    D in DEPARTMENTS
    where   D.College = 'Engineering';
```

Using the last query, there are three ways to specify iterator. (i) **D in DEPARTMENTS**, (ii)**DEPARTMENTS D**, and (iii) **DEPARTMENTS AS D**.

3.2.4 *Oracle: Object-Relational Database*

Because of its overall complexity the complete OQL standard has not yet been fully implemented in any software. But many of DBMS systems have given their own corresponding languages. This section presents the implementation of the object-relational school database specified in the UML class diagram shown in Figure 25. The implementation illustrates the use of user-defined types, reference types, and typed tables, as well as Oracle's support for collections in the form of variable-sized arrays (varrays) and nested tables, which is a feature that is not supported by the SQL standard. Object type hierarchies and a feature known as substitutable tables are used as a means to support class hierarchies [7].

```
 create type personUdt as
(   pld       varchar(11),
    firstName   varchar(20),
    lastName    varchar(20),
    dob         date)
instantiable not final ref is system generated;

create type facultyUdt under personUdt as
(   rank    varchar(10),
    advisorOf ref(campusClubUdt) scope campusClub array[5]
    references are checked on delete set null,
    worksln ref(departmentUdt) scope department
    references are checked on delete no action,
    chairOf ref(departmentUdt) scope department
    references are checked on delete set null)
instantiable not final;

create type studentUdt under personUdt as
(   status      varchar(10),
    memberOf    ref(campusClubUdt) scope campusClub array[5]
    references are checked on delete set null,
    major ref(departmentUdt) scope department
    references are checked on delete no action)
instantiable not final;

create table person of personUdt
(   primary key(pld),
    ref is personID system generated);

create  table faculty of facultyUdt under person;
create  table student of studentUdt under person;
create  type departmentUdt as
(   code    varchar(3),
    name    varchar(40),
    deptChair ref(facultyUdt) scope faculty
    references are checked on delete no action)
    instantiable not final ref is system generated
    method getStudents() returns studentUdt array[1000],
    method getFaculty() returns facultyUdt array[50]);

create table department of departmentUdt
(   primary key (code),
    deptChair with options not null,
    ref is departmentID system generated);

create type locationUdt as
(   street varchar(30),
    bldg varchar(5),
    room varchar(5)) not final;

create type campusClubUdt as
(   cld     varchar(10),
    name    varchar(50),
    location locationUdt,
    phone   varchar(12),
    advisor ref(facultyUdt) scope faculty
    references are checked on delete cascade,
    members ref(studentUdt) scope student array[50]
    references are checked on delete set null)
instantiable not final ref is system generated;

create table campusClub of campusClubUdt
(   primary key(cld),
    ref is campusClubID system generated);
```

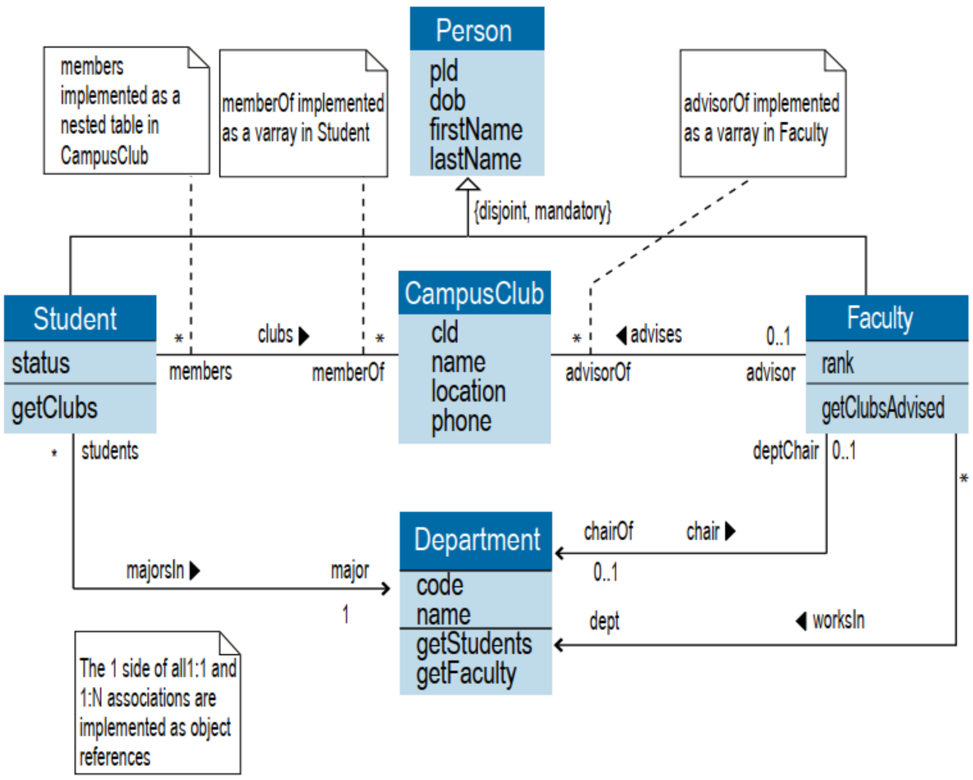


Figure 25: Class diagram for the Case Study of the School Database

Object types and type hierarchies

In conformance with the SQL standard, Oracle supports the basic object-relational feature of object types. An object type is a composite structure that can have a set of attributes and methods. Below is a simple example of a 'person_t' object type definition for the Person class without methods.

```
create or replace type person_t as object
(   pId         varchar2(9),
    firstName   varchar2(20),
    lastName    varchar2(20),
    dob         date);
```

To assign values to an object type, a variable can be created having the type of the object type. The following example illustrates how to assign values into the attributes of the person_t object type in PL/SQL.

```
declare
p   person_t;
begin
    p.pId := 'PR123456789';
    p.firstName := 'Terrence';
    p.lastName := 'Grand';
    p.dob := '10-NOV-1975';
end;
```

3.3 CONCLUSION

In this chapter, we explored the differences between object-oriented and relational databases, and discussed how modern databases have evolved to support both paradigms through the use of hybrid object-relational and object databases. We provided an example of Oracle database, which is a popular database management system that supports both relational and object-oriented data models.

We discussed the key features of object databases, such as support for complex data types, encapsulation, inheritance, and polymorphism, and highlighted their benefits in scenarios where the data is naturally hierarchical or complex. We also explored the limitations of relational databases in handling complex data types and relationships, and how object-oriented databases provide a more natural solution for such scenarios.

We then introduced the concept of hybrid object-relational databases, which combine the strengths of both relational and object-oriented databases to create a more versatile and flexible data management solution. We provided an example of how Oracle database supports hybrid models, allowing developers to define and store complex data types, such as nested tables and user-defined types, within the context of a relational database.

4 QUERYING XML DATABASES: XPATH AND XQUERY LANGUAGES

XML stands for eXtensible Markup Language. The use of XML has exploded in recent years. A huge amount of information is now stored in XML, both in XML databases and in documents on a file system. This includes highly structured data, such as sales figures, semi-structured data such as product catalogs and yellow pages. Even more information is transmitted between systems as transient XML documents [18].

4.1 XML DOCUMENT STRUCTURE

XML documents form a tree structure that starts at "the root" and branches to "the leaves". Figure 26 shows an example of hierarchical tree structure of an XML schema.

Example

The document bellow contains books in this XML representation. This representation is valid with the XML schema shown in Figure 26.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
    <book category="cooking">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="children">
        <title lang="en">Harry Potter</title>
```

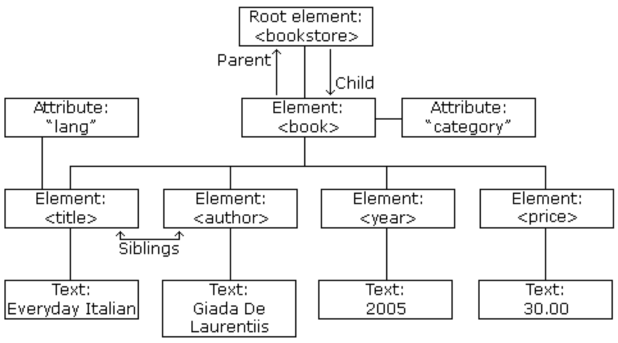


Figure 26: Tree structure of XML documents

```
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
    </book>
    <book category="web">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
</bookstore>
```

4.2 QUERYING XML DOCUMENTS

We assume that it is necessary to convert XML data to some other representation (say, relational) and query that converted data using some language (such as SQL). Sometimes that is the most appropriate strategy - for example, if the XML data is highly regular and will be queried many times in the same way, you may be able to query it more efficiently in a purely relational context. Often, though, you want to store and represent the data as XML throughout its life (or at least preserve the XML abstraction over your data when querying) [13].

Querying XML data is different from querying relational data - it requires navigating around a tree structure that may or may not be well defined. XQuery is an SQL-like language designed by the W3C to address these needs. It allows you to select the XML data elements of interest, reorganize and possibly transform them, and return the results in a structure of your choosing [18].

4.2.1 *XPath and XQuery languages*

XPath is a language for selecting elements and attributes from an XML document while traversing its hierarchy and filtering out unwanted content. XPath 1.0 is a useful recommendation that specifies path expressions and a limited set of functions. XPath 2.0 has become much more than that, encompassing a wide variety of expressions and functions, not just path expressions. XQuery 1.0 and XPath 2.0 overlap to a very large degree. They have the same data model and the same set of built-in functions and operators. XQuery has a number of features that are not included in XPath, such as FLWORs (standing for *For, Let, Where, and Order By* clauses) and XML constructors. This is because these features are not relevant to selecting, but instead have to do with structuring or sorting query results.

4.2.2 *Processing Queries*

A simple example of a processing model for XQuery is shown in Figure 27. This section describes the various components of this model.

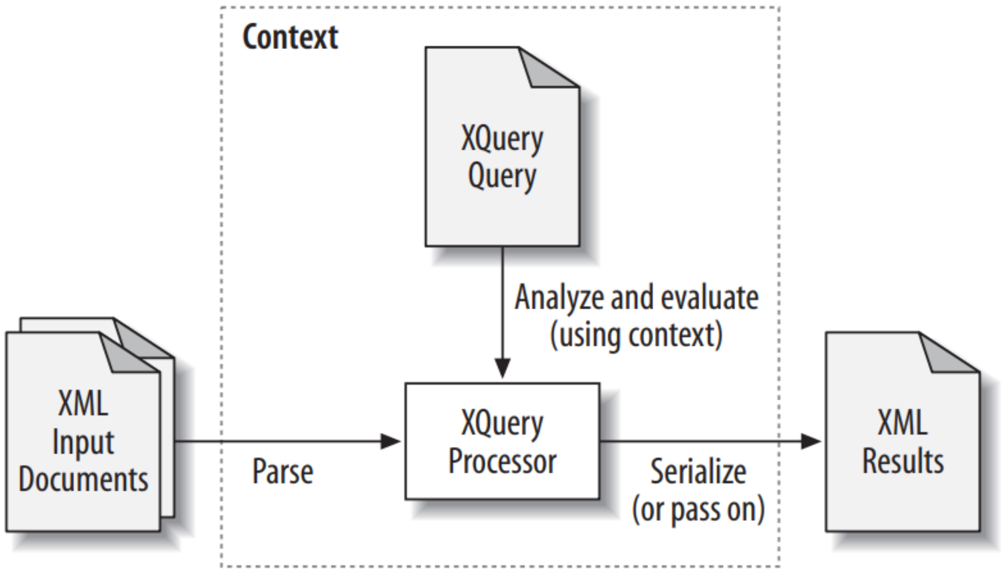


Figure 27: XQuery processor basis

4.3 XPATH

The XML Path Language - XPath, as it is more commonly known - was first published as a recommendation by the W3C in 1999. XPath is a language for addressing parts of XML documents. Querying facilities in general function to locate or identify certain information. Because querying facilities in general function to locate or identify certain information, it's easy to see that XPath is itself a sort of query language [13].

4.3.1 *XPath expression*

The typical XPath expression deliberately uses a "path like" syntax similar to that notation used by operating systems for referencing files and directory paths. A *path expression* is used by XPath to identify and navigate nodes in an XML document. And as a result, it evaluates to a value that has one of four possible types: node set (an ordered collection of nodes), a Boolean value, a number, or a string.

There are many different kind of expressions. The most important are : (i) *path expression*, (ii) *value expression*, and (iii) *node set expression*.

4.3.1.1 *Path expression*

A path expression looks sometimes like this:

/book/title[@lang="en"]

The execution of the above expression returns all books written in english from all those in the book catalog described in Section 4.1. To select nodes, we can also pass a set of parameters, logic expressions, and/or make call for many kinds of functions. The table below listed some XPath expressions and the result of the expressions:

XPath Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Table 8: Querying book store document

4.3.1.2 *Value expression*

The value expressions include:

- String and numeric literals Variable references
- Function invocations
- Logical expressions ("and" and "or")
- Comparison expressions (=, <, >, <=, >=, and .'=)
- Arithmetic expressions (+, -, *, d i v [because / is used for other purposes in path expressions], and mod)

4.3.1.3 *Node set expression*

Node set expressions (|, pronounced "union" - combines two node sets into one).

4.4 XQUERY

Like the SQL dedicated to relational databases, XQuery is intended to be a nonprocedural language that gives facilities for querying data stored in multiple XML documents. XQuery is designed to meet the requirements identified by the W3C XML Query Working Group [XML Query 1.0 Requirements] and the use cases in [XML Query Use Cases]. It is designed to be a language in which queries are concise and easily understood. It is

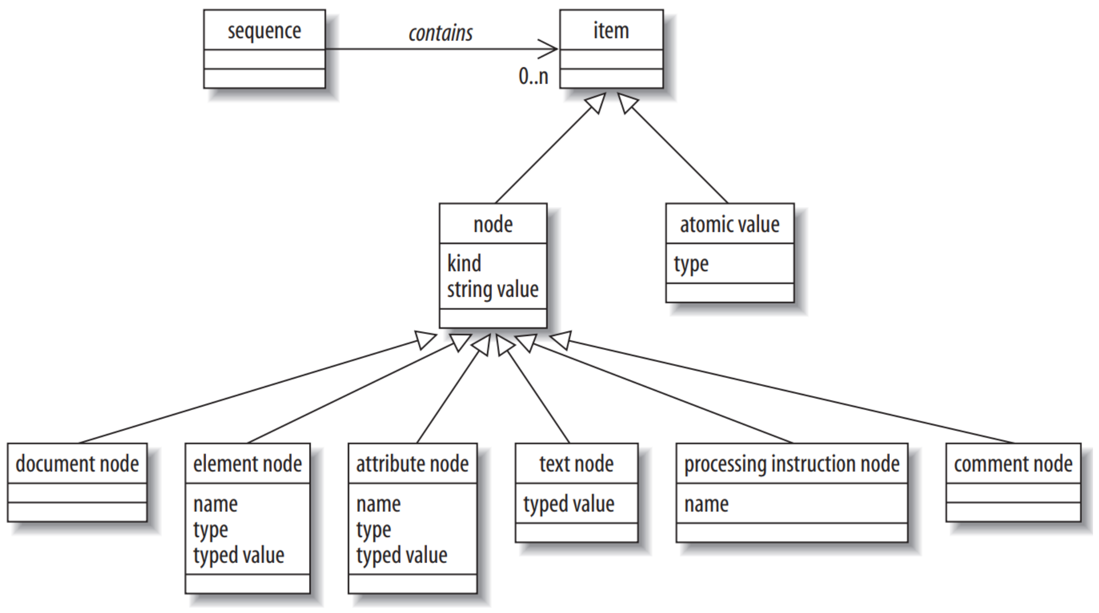


Figure 28: Basic components of the data model

also flexible enough to query a broad spectrum of XML information sources, including both databases and documents.

4.4.1 *The XQuery Data Model*

XQuery has a data model that is used to define formally all the values used within queries, including those from the input document(s), those in the results, and any intermediate values. The XQuery data model is officially known as the XQuery 1.0 and XPath 2.0 Data Model, or XDM.

Understanding the XQuery data model is analogous to understanding tables, columns, and rows when learning SQL. It describes the structure of both the inputs and outputs of the query. The basic components of XDM are:

- **Node:** An XML construct such as an element or attribute
- **Atomic value:** A simple data value with no markup associated with it.
- **Item:** A generic term that refers to either a node or an atomic value
- **Sequence:** An ordered list of zero, one, or more items.

The relationship among these components is depicted in Figure 28.

Nodes

Nodes are used to represent XML constructs such as elements and attributes. For example, the path expression doc("catalog.xml")/catalog/product returns four product element nodes.

XQuery uses six kinds of nodes:

1. *Element nodes:* Represent an XML element
2. *Attribute nodes:* Represent an XML attribute
3. *Document nodes:* Represent an entire XML document (not its outermost element)
4. *Text nodes:* Represent some character data content of an element
5. *Processing instruction nodes:* Represent an XML processing instruction
6. *Comment nodes:* Represent an XML

Example

When translating the following small XML document to the XQuery data model, it looks like the diagram in Figure 29

```
<catalog xmlns="http://datypic.com/cat">
    <product dept="MEN" xmlns="http://datypic.com/prod">
        <number>784</number>
        <name language="en">Cotton Dress Shirt</name>
        <colorChoices>white gray</colorChoices>
        <desc>Our <i>favorite</i> shirt!</desc>
    </product>
</catalog>
```

Root node

XPath 1.0 has a separate concept of a *root node*, which is equivalent to a document node in XQuery (and XPath 2.0). A *root node* represents the entire document and would be the parent of the catalog element in our previous example.

Node identity and name

Every node has a unique identity. You may have two XML elements in the input document that contain the exact same data, but that does not mean they have the same identity.

String and typed values of nodes

There are two kinds of values for a node: string and typed. The string value of an element node is its character data content and that of all its descendant elements concatenated together. The string value of an attribute node is simply the attribute value. The string value of a node can be accessed using the string function. For example:

```
string(doc("catalog.xml")/catalog/product[4]/number)
```

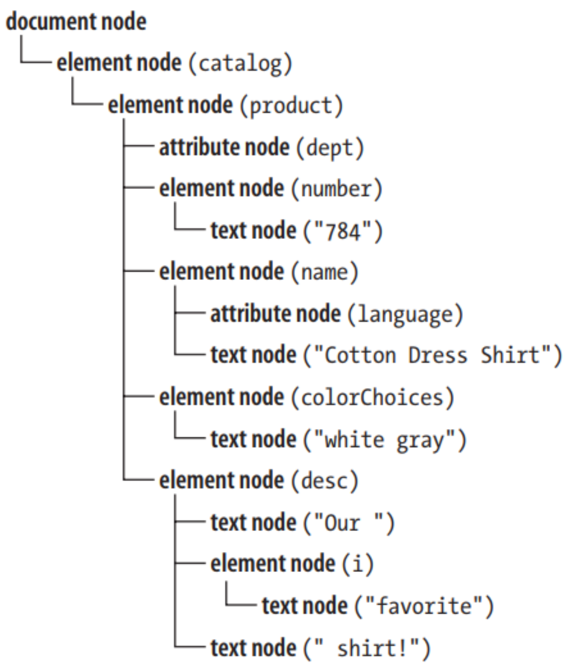


Figure 29: Node hierarchy

returns the string 784, while:

```
string(<desc>Our <i>favorite</i> shirt!</desc>)
```

returns the string Our favorite shirt!, without the i start and end tags.

An element or attribute might have a particular type if it has been validated with a *schema*. The typed value of a node can be accessed using the data function. For example:

```
data(doc("catalog.xml")/catalog/product[4]/number)
```

returns the integer 784, if the number element is declared in a schema to be an integer. If it is not declared in the schema, its typed value is still 784, but the value is considered to be *untyped*.

Atomic Values

An atomic value is a simple data value such as 784 or ACC, with no markup, and no association with any particular element or attribute. An atomic value can have a specific type, such as xs:integer or xs:string, or it can be untyped.

Sequences

Sequences are ordered collections of items. A sequence can contain zero, one, or many items. Each item in a sequence can be either an atomic value or a node. For example, the expression doc("catalog.xml")/catalog/product returns a sequence of four items, which happen to be product element nodes.

A sequence can also be created explicitly using a sequence constructor. The syntax of a sequence constructor is a series of values, delimited by commas, surrounded by parentheses. For example, the expression (1, 2, 3) creates a sequence consisting of those three atomic values.

Some of the most used functions on sequences are the aggregation functions (min, max, avg, sum). In addition, union, except, and intersect expressions allow sequences to be combined. There are also a number of functions that operate generically on any sequence, such as index-of and insert-before.

Types

The XQuery type system is based on that of XML Schema. XML Schema has built-in simple types representing common datatypes such as **xs:integer**, **xs:string**, and **xs: date**.

4.4.2 *Literals and Variables*

Literals

Literals are simply constant values that are directly represented in a query, such as "ACC" and 29.99. There are two kinds of literals: *string literals* and *numeric literals*. They can be used in expressions anywhere a constant value is needed, for example the strings in the comparison expression: if ($department = "ACC") then "accessories" else "other" or the numbers 1 and 30 in the function call: substring($name, 1, 30)

Variables

Variables in XQuery are identified by names that are preceded by a dollar sign **($).***
Example 1, declare an initialized variable $num.

```
declare variable $num := 12;
```

Example 2, to return the maximum of a sequence (1, 2, 3, 4) we declared in the following example two variables $seq and $m.

```
let $seq := (1,2,3,4)
let $max := max($seq)
return $seq
```

4.4.3 *Categories of XQuery Expressions*

The categories of expressions available are summarized in Table 9.

4.5 FLOWR EXPRESSIONS

The heart of XQuery is in five clauses (**F**or, **L**et, **O**rder By, **W**here, and **R**eturn). These fives clauses are abbreviated in the term FLWOR and pronounced *Flower*. The presence of these clause in a query follows a set of rules

Category	**Description**	**Operators or keywords**
Primary	The basics: literals, variables, function calls, and parenthesized expressions	
Comparison	Comparison based on value, node identity, or document order	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge, is, «, »
Conditional	If-then-else expressions	if, then, else
Logical	Boolean and/or operators	or, and
Path	Selecting nodes from XML documents	/, //, .., ., child::, etc
Constructor	Adding XML to the results	<, >, element, attribute
FLWOR	Controlling the selection and processing of nodes	for, let, where, order by, return
Quantified	Determining whether sequences fulfill specific conditions	some, every, in, satisfies
Sequence-related	Creating and combining sequences	to, union (|), intersect, except
Type-related	Casting and validating values based on type	instance of, typeswitch, cast as, castable, treat, validate
Arithmetic	Adding, subtracting, multiplying, and dividing	+, -, *, div, idiv, mod

Table 9: Categories of expressions

1. The five clauses should be written in lower case.
2. The query should start either by **“for”** or **“let“** clauses.
3. **”where”** and **“order by”** clauses can not be used if **“for”** clause is not used in the query.
4. The query should be finished by **“return”** clause.

4.5.1 *Case study: Catalog.xml and price.xml documents*

The following document “catalog.xml” represents a catalog of a set of products belonging each one to a department represented by the attribute `dept` and having a number, a name given in such `language`, a color represented by the element `colorChoices` and having sometimes a description (see the last product in this document). We use later another document “price.xml” where a list of prices are given for a set of products in different dates.

```
<catalog>
    <product dept="WMN">
        <number>557</number>
        <name language="en">Fleece Pullover</name>
        <colorChoices>navy black</colorChoices>
    </product>
    <product dept="ACC">
        <number>563</number>
        <name language="en">Floppy Sun Hat</name>
    </product>
    <product dept="ACC">
        <number>443</number>
        <name language="en">Deluxe Travel Bag</name>
    </product>
    <product dept="MEN">
        <number>784</number>
        <name language="en">Cotton Dress Shirt</name>
        <colorChoices>white gray</colorChoices>
        <desc>Our <i>favorite</i> shirt!</desc>
    </product>
</catalog>
```

The previous document will be used in the following sections.

4.5.2 *Return all elements of a selected type*

The following query returns all products.

```
for $p in doc("catalog.xml")/catalog/product
return $p
```

Because the hole document contains one king of product element, the following query is equivalent to the previous one.

```
for $p in doc("catalog.xml")//product
return $p
```

4.5.3 *Return all elements that have a specific child element or attribute*

The two following queries returns respectively all products that have **desc** element and **dept** attribute.

```
Example 1.  for $p in doc("catalog.xml")/catalog/product[desc]
            return $p

Example 2.  for $p in doc("catalog.xml")/catalog/product[@dept]
            return $p
```

If we want retrieve the products that have dept = "ACC", we can modify the last query (Example 2) as follow

```
Example 3.  for $p in doc("catalog.xml")/catalog/product[@dept="ACC"]
            return $p
```

The queries of the remain sections are given to query the document "price.xml". This document contains the price lists of a set of products seen in the document "catalog.xml". When the price of some products change, we keep their new prices in this document.

```
<prices>
    <priceList effDate="2006-11-15">
        <prod num="557">
            <price currency="USD">29.99</price>
            <discount type="CLR">10.00</discount>
        </prod>
        <prod num="563">
            <price currency="USD">69.99</price>
            <discount type="CLR">10.00</discount>
        </prod>
        <prod num="443">
            <price currency="USD">39.99</price>
            <discount type="CLR">3.99</discount>
        </prod>
    </priceList>
    <priceList effDate="2009-01-01">
        <prod num="557">
```

```
            <price currency="USD">40.99</price>
            <discount type="CLR">10.00</discount>
        </prod>
    </priceList>
</prices>
```

4.5.4 *Example of complete FLWOR expression*

The following query returns all the prices of the product number 557 ordered by price value. We intentionally used all five clauses FLWOR in this query.

```
for $p in doc("price.xml")//prod
let $num := $p/@num
let $price := $p/price
where $num = 557
order by $price
return $p
```

4.5.5 *XML elements construction by XQuery*

We can construct XML elements in the output of an XQuery script. In the following example, we want return the prices of all product constructed with the dates of changing.

```
for $p in doc("price.xml")//prod
let $a := string($p/price),
    $d := string($p/parent::priceList/@effDate)
return <price date="{$d}">{$a}</price>
```

The output of this query is given as follow

```
<price date="2006-11-15">29.99</price>
<price date="2006-11-15">69.99</price>
<price date="2006-11-15">39.99</price>
<price date="2022-01-01">40.99</price>
```

The function **string()** in **string($p/price)** and **string($p/parent::priceList/@effDate)** is used to get the values of the **price** element and **effDate** attribute respectively.

4.6 conclusion

In this chapter, we introduced XPath and XQuery, two powerful query languages used for querying and transforming XML documents. XPath is a language for navigating through the hierarchical structure of an XML document and selecting specific elements or attributes. XQuery is a more expressive language that builds on XPath and allows for more complex queries and transformations.

We discussed the syntax and semantics of both languages, and how they can be used to extract data from XML documents and transform them into different formats. We also highlighted the similarities and differences between XPath and XQuery, and when to use each language based on the complexity of the task at hand.

We explored the key features of both languages, such as their support for a wide range of data types, functions, and operators, and their ability to handle complex queries and transformations. We provided examples of how to use XPath and XQuery to query and transform XML documents, including selecting specific elements or attributes, filtering results, and performing mathematical and logical operations.

Overall, this chapter provides an overview of XPath and XQuery, two essential query languages for working with XML documents. By understanding the syntax and semantics of these languages, readers will be better equipped to extract and transform data from XML documents and integrate them with other systems and applications.

5 DISTRIBUTED DATABASES

5.1 INTRODUCTION

Distributed database system (DDBS) technology is the union of two approaches to data processing: database system and computer network technologies. *Database systems* have taken us from a paradigm of data processing in which each application defined and maintained its own data to one in which the data are defined and administered centrally. This new orientation results in data independence, whereby the application programs are immune to changes in the logical or physical organization of the data, and vice versa. The technology of *computer networks*, on the other hand, promotes a mode of work that goes against all centralization efforts. At first glance it might be difficult to understand how these two contrasting approaches can possibly be synthesized to produce a technology that is more powerful and more promising than either one alone.

5.1.1 *Distributed Data Processing*

Distributed data processing is a computing model in which data processing is distributed across multiple computers or nodes in a network. The processing can be done in parallel, allowing for faster and more efficient processing of large amounts of data. Each node in the network has access to a subset of the data, and the nodes work together to process the data and generate the desired output.

5.1.2 *Distributed Database System*

A distributed database system is a type of database system that is spread across multiple computers geographically distributed. In a distributed database system, the data is partitioned or replicated across multiple nodes, and the nodes work together to process queries and transactions from clients.

A DDBS is also not a system where, despite the existence of a network, the database resides at only one node of the network. What we are interested in is an environment where data are distributed among a number of sites Figure 30.

Several benefits are given when using distributed database systems:

- Scalability: Distributed database systems can scale horizontally by adding more nodes to the network. This allows the system to handle large volumes of data and high transaction rates.

- Fault tolerance: Distributed database systems can continue to operate even if one or more nodes fail. Data can be replicated across multiple nodes, so if one node fails, another node can take over without loss of data.

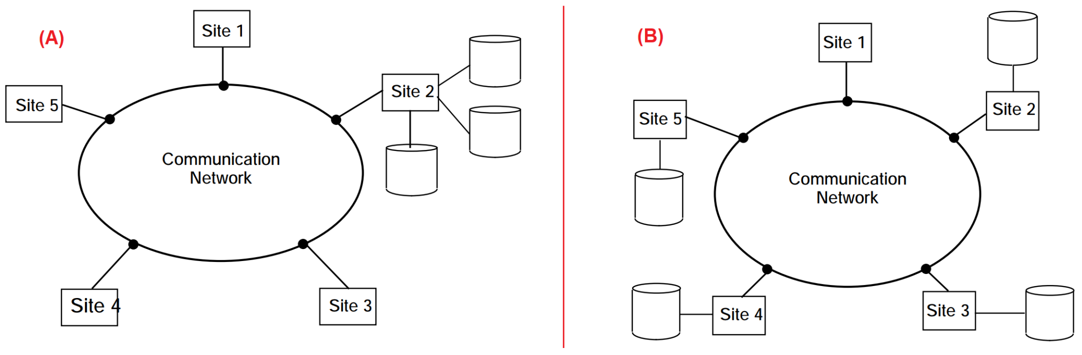


Figure 30: Centralized vs Distributed Database Systems: **(A)** Centralized Database System on network, **(B)** DDBS environment.

- Improved performance: By distributing the data and processing across multiple nodes, distributed database systems can improve performance by processing queries and transactions in parallel.

5.2 DISTRIBUTED DBMS ARCHITECTURE

The architecture of a system defines its structure. This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined. The specification of the architecture of a system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system.

Before going to present the architecture of a distributed DBMS we need firstly to present the "ANSI/SPARC Architecture"

5.2.1 *ANSI/SPARC Architecture*

ANSI/SPARC Architecture is an early milestone in the field of database systems. It was developed by the American National Standards Institute (ANSI) and the Standards Planning and Requirements Committee (SPARC) in the 1970s, when the field of database management was still in its early stages, and it helped to establish many of the fundamental concepts and principles that are still used today.

The ANSI/SPARC architecture defines three levels of abstraction for a database system (Figure 31):

- External level: This is the level of the database system that is visible to end-users and applications. It describes how data is viewed by different users and groups, and how data is accessed and manipulated by applications. Each external schema is tailored to meet the specific needs of a particular user or application.

- Conceptual level: This is the level of the database system that describes the overall logical structure of the database. It defines the relationships between different types

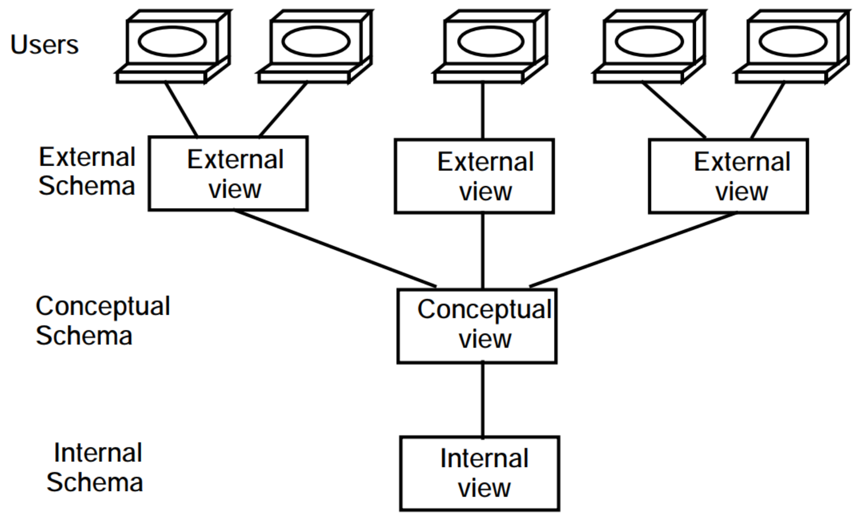


Figure 31: ANSI/SPARC Architecture

of data and how they are organized and stored in the database. The conceptual schema is independent of any particular application or user, and is used to ensure that all data in the database is consistent and integrated.

- Internal level: This is the level of the database system that describes how data is physically stored and accessed by the computer system. It defines the storage structures and access methods used by the DBMS to manage the data. The internal schema is hidden from users and applications, and is optimized for efficient storage and retrieval of data.

5.2.2 *Architectural Models for Distributed DBMSs*

The ways in which a distributed DBMS can be architected can be classified in terms of: (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity. (see Figure 32).

5.2.2.1 *Autonomy*

Autonomy, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Requirements of an autonomous system have been specified as follows [10].

- The local operations of the individual DBMSs are not affected by their participation in the distributed system.

- The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.

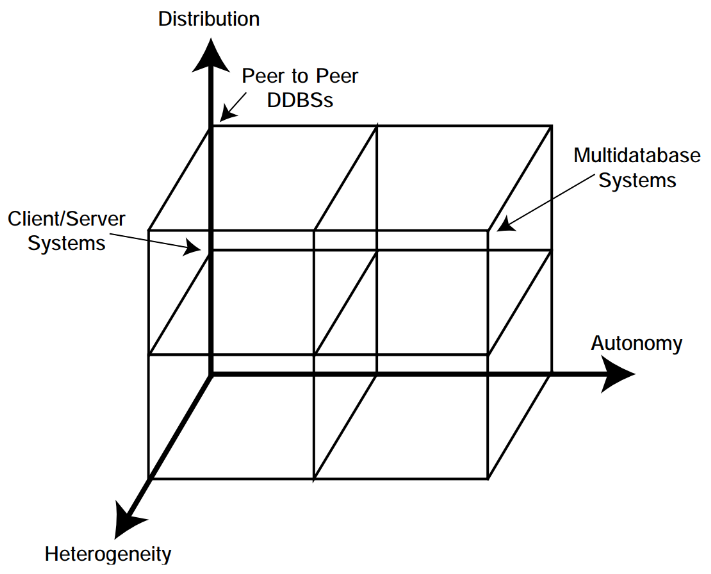


Figure 32: DBMS Implementation Alternatives

- System consistency or operation should not be compromised when individual DBMSs join or leave the distributed system.

On the other hand, the dimensions of autonomy can be specified as follows [8]

- Design autonomy: Individual DBMSs are free to use the data models and transaction management techniques that they prefer.
- Communication autonomy: Each of the individual DBMSs is free to make its own decision as to what type of information it wants to provide to the other DBMSs or to the software that controls their global execution.
- Execution autonomy: Each DBMS can execute the transactions that are submitted to it in any way that it wants to.

5.2.2.2 *Distribution*

The distribution dimension of the taxonomy deals with data. We are considering the physical distribution of data over multiple sites; as we discussed earlier, the user sees the data as one logical pool. There are a number of ways DBMSs have been distributed. We abstract these alternatives into two classes: *client/server* distribution and *peer-to-peer* distribution (or *full distribution*).

In the context of distributed databases, the difference between client/server distribution and peer-to-peer distribution is as follows:

Client/server distribution of a distributed database involves a centralized server that manages and coordinates the database, while the clients are responsible for accessing and querying the data. The server is responsible for maintaining the consistency and integrity of the data by ensuring that all clients see the same version of the data at any given time.

This approach is often used when there is a large amount of data that needs to be shared among multiple clients. On the other hand, in peer-to-peer distribution of a distributed database, all participants in the network have equal status, and each participant can act as both a client and a server. Each participant maintains a local copy of the data and shares it with other participants in the network. This approach is often used in situations where there is no centralized authority, and where participants need to share data with each other without relying on a central server.

5.2.2.3 *Heterogeneity*

Heterogeneity refers to the presence of diversity or differences in a distributed database environment in terms of data models, query languages, and transaction management protocols. In terms of data models, heterogeneity can arise when different databases in the distributed environment use different data models. For example, some databases may use a relational data model, while others may use a hierarchical or network data model. This can create challenges in integrating data from different sources and querying across multiple databases. Similarly, heterogeneity in SQL standard relational query language, there are many different implementations and every vendors language has a slightly different flavor (sometimes even different semantics, producing different results). Heterogeneity in query languages can also arise when different databases use different query languages to access and manipulate data. For example, some databases may use SQL, while others may use a NoSQL query language. This can make it difficult to write queries that work across multiple databases.

5.2.3 *Client/Server architecture*

Client/server DBMSs entered the computing scene at the beginning of 1990s and have made a significant impact on both the DBMS technology and the way we do computing. The general idea is very simple and elegant: distinguish the functionality that needs to be provided and divide these functions into two classes: server functions and client functions. This provides a two-level architecture which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution. We can cite many examples of DDBMS that use client/server architecture of distributed database systems. One such example is Microsoft SQL Server, Oracle Database, MySQL and PostgreSQL. Figure 33 depicts the architecture of Client/Server DDB system.

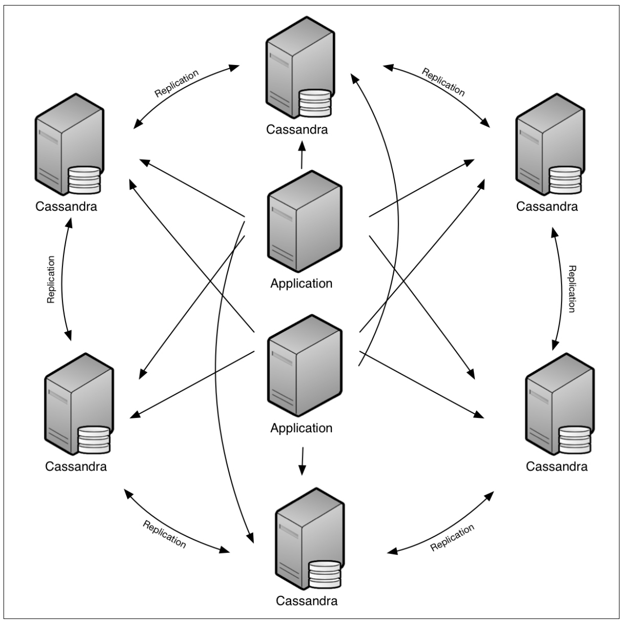


Figure 34: Apache cassandra peer-to-peer DDBMS

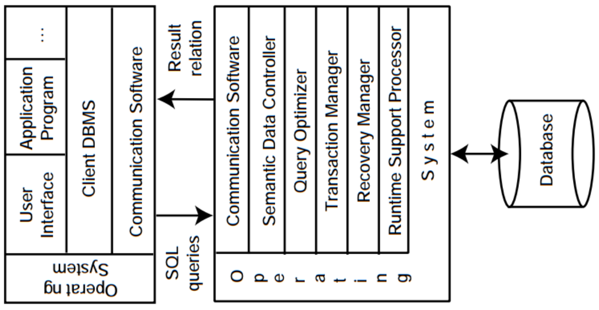


Figure 33: Client/Server Reference Architecture

5.2.4 *Peer-To-peer Architecture*

After a decade of popularity of client/server computing, peer-to-peer have made a come-back in the last few years (primarily spurred by file sharing applications) and some have even positioned peer-to-peer data management as an alternative to distributed DBMSs. Apache Casandra DBMS represent a good example of peer-to-peer DDBMS [1]. Unlike either monolithic or master-slave designs, Cassandra makes use of an entirely peer-to-peer architecture. All nodes in a Cassandra cluster can accept reads and writes, no matter where the data being written or requested actually belongs in the cluster Figure 34 [17].

[1]https://cassandra.apache.org/

5.2.5 *Distributed query processing*

Distributed query processing is the process of executing a database query that involves data stored on multiple nodes or servers in a distributed database system. When a query is submitted, it must be broken down into smaller subqueries that can be executed on different nodes in parallel, and then the results must be combined to form the final result set. Distributed query processing involves several steps, including query optimization, query decomposition, data fragmentation and distribution, data transfer, local processing, and result consolidation. The goal of distributed query processing is to minimize the amount of data that needs to be transferred between nodes and to maximize parallelism in the execution of subqueries, in order to improve query performance and scalability in a distributed environment. It is a key challenge in designing and implementing distributed database systems.

Because it is a critical performance issue, query processing has received (and continues to receive) considerable attention in the context of both centralized and distributed DBMSs. However, the query processing problem is much more difficult in distributed environments than in centralized ones, because a larger number of parameters affect the performance of distributed queries. In particular, the relations involved in a distributed query may be fragmented and/or replicated, thereby inducing communication overhead costs. Furthermore, with many sites to access, query response time may become very high.

5.2.5.1 *Query processing problem*

The main function of a *relational query processor* is to transform a high-level query (typically, in relational calculus) into an equivalent lower-level query (typically, in some variation of relational algebra). The low-level query actually implements the execution strategy for the query. The transformation must achieve both *correctness* and *efficiency*. It is correct if the low-level query has the same semantics as the original query, that is, if both queries produce the same result. Since each equivalent execution strategy can lead to very different consumptions of computer resources, the main difficulty is to select the execution strategy that minimizes resource consumption.

Example

We consider the following subset of the engineering database schema given in Figure 37.

```
EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)
```

and the following simple user query: "Find the names of employees who are managing a project".

The expression of the query in relational calculus using the SQL syntax is

```
SELECT ENAME
FROM EMP,ASG
```

```
WHERE EMP.ENO = ASG.ENO
AND RESP = "Manager"
```

Two equivalent relational algebra queries that are correct transformations of the query above are:

$\Pi_{NAME}(\sigma_{RESP=''Manager'' \wedge EMP.ENO}(EMP \times ASG))$, and

$\Pi_{NAME}(EMP \bowtie_{ENO} (\sigma_{RESP=''Manager''}(ASG)))$

It is intuitively obvious that the second query, which avoids the Cartesian product of EMP and ASG, consumes much less computing resources than the first, and thus should be retained. In a centralized context, query execution strategies can be well expressed in an extension of relational algebra. The main role of a centralized query processor is to choose, for a given query, the best relational algebra query among all equivalent ones. In a distributed system, relational algebra is not enough to express execution strategies. It must be supplemented with operators for exchanging data between sites. Besides the choice of ordering relational algebra operators, the distributed query processor must also select the best sites to process data, and possibly the way data should be transformed. This increases the solution space from which to choose the distributed execution strategy, making distributed query processing significantly more difficult.

Example

We consider the following query:

$\Pi_{NAME}(EMP \bowtie_{ENO} (\sigma_{RESP=''Manager''}(ASG)))$

We assume that relations EMP and ASG are horizontally fragmented as follows:

$EMP_1 = \sigma_{ENO \leqslant ''E3''}(EMP)$

$EMP_2 = \sigma_{ENO > ''E3''}(EMP)$

$ASG_1 = \sigma_{ENO \leqslant ''E3''}(ASG)$

$ASG_2 = \sigma_{ENO > ''E3''}(ASG)$

Fragments ASG_1, ASG_2, EMP_1, and EMP_2 are stored at sites 1, 2, 3, and 4, respectively, and the result is expected at site 5. For the sake of pedagogical simplicity, we ignore the project operator in the following. Two equivalent distributed execution strategies for the above query are shown in Figure 35

5.3 DISTRIBUTED DATABASE DESIGN

In the design of a distributed DBMSs, the distribution of applications involves two things: the distribution of the distributed DBMS software and the distribution of the application programs that run on it. Two major strategies that have been identified for designing distributed databases are the top-down approach and the bottom-up approach.

5.3.1 *Top-down approach*

The activity of top-down approach designs (Figure 36) with a requirements analysis that defines the environment of the system and "elicits both the data and processing needs of all potential database users". The requirements study also specifies where the final

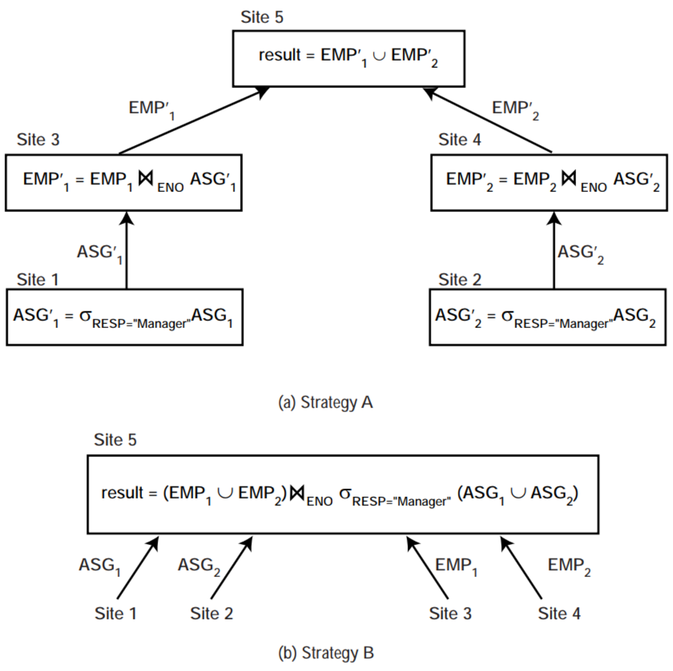


Figure 35: Equivalent Distributed Execution Strategies

system is expected to stand with respect to the objectives of a distributed DBMS. These objectives are defined with respect to performance, reliability and availability, economics, and expandability (flexibility).

The requirements document is input to two parallel activities: *view design* and *conceptual design*. The view design activity deals with defining the interfaces for end users. The conceptual design, on the other hand, is the process by which the enterprise is examined to determine entity types and relationships among these entities. The global conceptual schema (GCS) and access pattern information collected as a result of view design are inputs to the distribution design step. The objective at this stage is to design the local conceptual schemas (LCSs) by distributing the entities over the sites of the distributed system.

The relation in distributed databases can be divided into sub-relations called fragments. Thus, the distribution design activity consists of two steps: *fragmentation* and *allocation*.

5.3.2 *Fragmentation Alternatives*

Fragmentation in distributed databases occurs when data is divided into smaller subsets and distributed across multiple nodes in a network. Relation instances are essentially tables, so the issue is one of finding alternative ways of dividing a table into smaller ones. There are clearly two alternatives for this: dividing it *horizontally* or dividing it *vertically*.

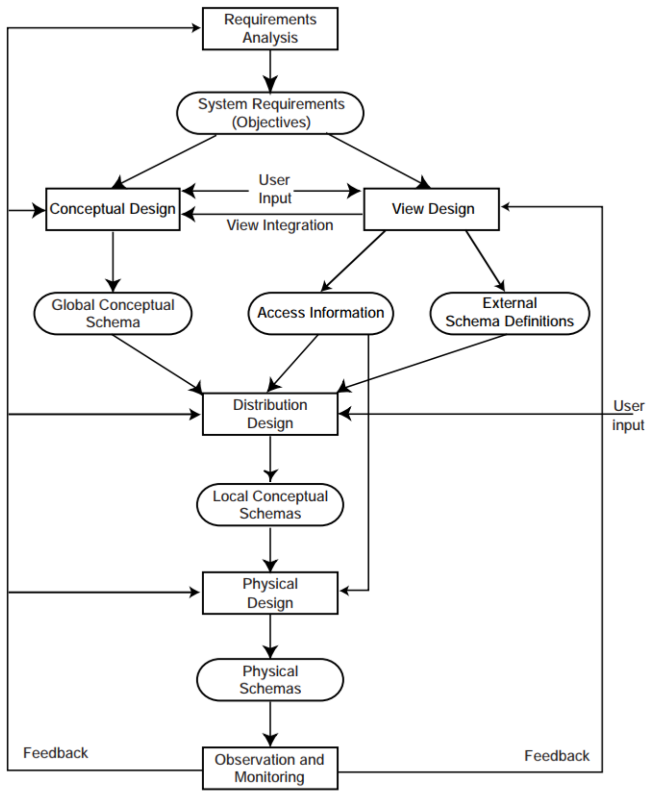
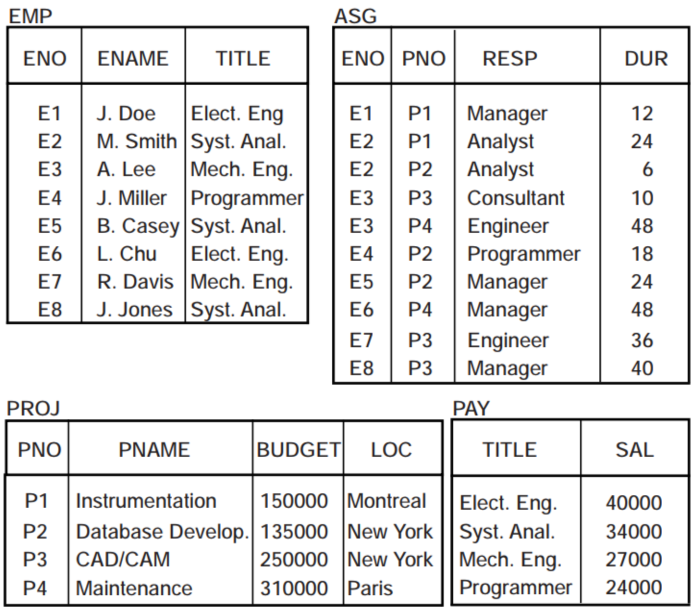


Figure 36: Top-down design process



EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Figure 37: Relational schema. Set of employees (**EMP**) assigned (**ASG**) to projects (**PROJ**) for different payments (**PAY**)

.

Example

Figure 38 shows the PROJ relation of Figure 37 divided horizontally into two relations. Subrelation PROJ1 contains information about projects whose budgets are less than $200,000, whereas PROJ2 stores information about projects with larger budgets.

5.3.3 *Correctness Rules of Fragmentation*

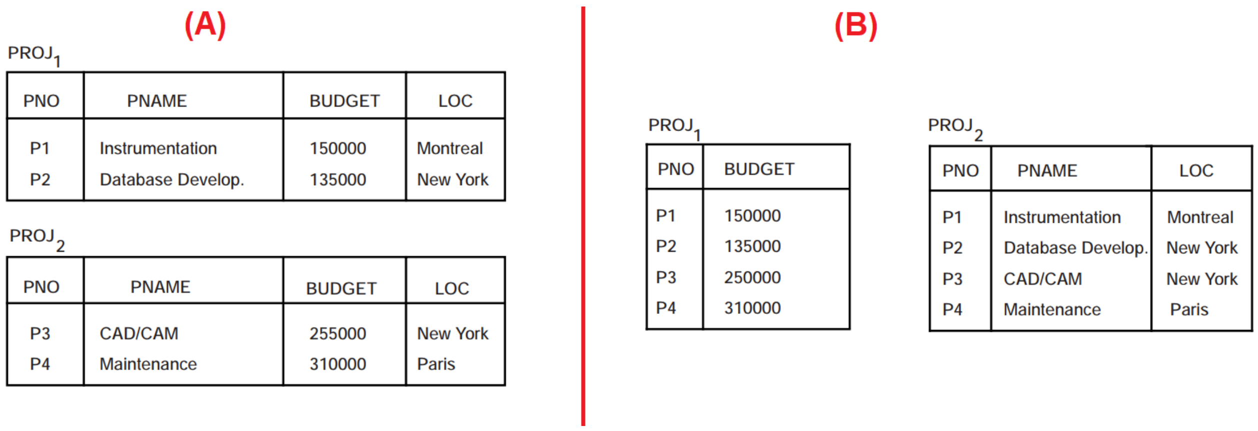
We will enforce the following three rules during fragmentation, which, together, ensure that the database does not undergo semantic change during fragmentation.

1. *Completeness*

If a relation instance R is decomposed into fragments $F_R = \{R_1, R_2, ..., R_n\}$, each data item that can be found in R can also be found in one or more of R_i's. This property, which is identical to the *lossless* decomposition property of normalization, is also important in fragmentation since it ensures that the data in a global relation are mapped into fragments without any loss. Note that in the case of horizontal fragmentation, the "item" typically refers to a tuple, while in the case of vertical fragmentation, it refers to an attribute.

2. *Reconstruction*

If a relation R is decomposed into fragments $F_R = \{R_1, R_2, ..., R_n\}$, it should be possible to define a relational operator ∇ such that $R = \nabla R_i, \forall R_i \in F_R$. The operator ∇ will be differ-



(A)

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

(B)

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Figure 38: (**A**) Horizontal fragmentation of the relation PRJ. Whereas (**B**) is a vertical fragmentation of the relation PRJ

.

ent for different forms of fragmentation; it is important, however, that it can be identified. The reconstructability of the relation from its fragments ensures that constraints defined on the data in the form of dependencies are preserved.

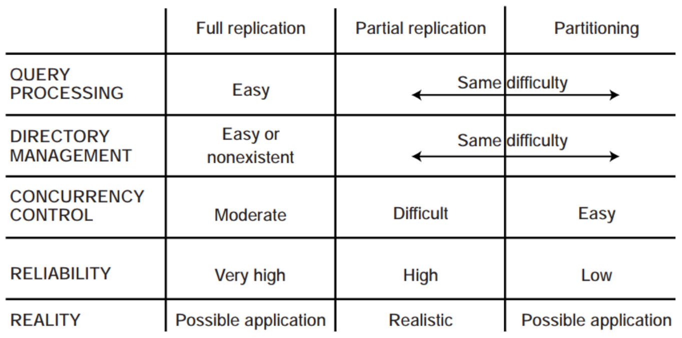
3. *Disjointness*

If a relation R is horizontally decomposed into fragments $F_R = \{R_1, R_2, ..., R_n\}$ and data item d_i is in R_j, it is not in any other fragment $R_k (k \neq j)$. This criterion ensures that the horizontal fragments are disjoint. If relation R is vertically decomposed, its primary key attributes are typically repeated in all its fragments (for reconstruction). Therefore, in case of vertical partitioning, disjointness is defined only on the non-primary key attributes of a relation.

5.3.4 *Data allocation*

If the database is properly fragmented, the fragments can be stored on different sites on the network, and the data can either be replicated or maintained as a single copy. Replication helps to ensure reliability and the efficiency of read-only queries, as it increases the chances of accessible data even in the event of system failures. This also allows for parallel execution of read-only queries that access the same data items, as multiple copies exist on multiple sites. On the other hand, the execution of update queries cause trouble since the system has to ensure that all the copies of the data are updated properly. Hence the decision regarding replication is a trade-off that depends on the ratio of the read-only queries to the update queries.

A non-replicated database (commonly called a *partitioned* database) contains fragments that are allocated to sites, and there is only one copy of any fragment on the network. In case of replication, either the database exists in its entirety at each site (fully replicated database), or fragments are distributed to the sites in such a way that copies of a fragment may reside in multiple sites (partially replicated database). In the latter the number of



	Full replication	Partial replication	Partitioning
QUERY PROCESSING	Easy	Same difficulty	
DIRECTORY MANAGEMENT	Easy or nonexistent	Same difficulty	
CONCURRENCY CONTROL	Moderate	Difficult	Easy
RELIABILITY	Very high	High	Low
REALITY	Possible application	Realistic	Possible application

Figure 39: Replication alternatives

copies of a fragment may be an input to the allocation algorithm or a decision variable whose value is determined by the algorithm. Figure 39 provides a comparison between replication alternatives.

5.4 CONCLUSION

In this chapter, we discussed the concept of distributed databases, which are designed to manage data across multiple physical locations and provide a more scalable and fault-tolerant data management solution. We explored the key concepts and technologies involved in distributed data processing, including distributed data systems, distributed DBMS architecture, and distributed data design.

We highlighted the benefits of distributed databases, including improved scalability, increased availability, and better performance for large-scale data processing. We also discussed the challenges of designing and implementing distributed databases, such as data consistency, network latency, and system complexity.

We examined the different types of distributed database architectures, such as client-server, peer-to-peer, and hybrid models, and how they can be used to balance performance, scalability, and fault tolerance. We also discussed the importance of distributed data design.

BIBLIOGRAPHY

[1] ARMSTRONG, W. W. (1974). Dependency structures of database relationship. *Information Processing*, pages 580–583.

[2] Burleson, D. K. (1998). *Inside the Database Object Model*. CRC Press.

[3] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

[4] Date, C. (2003). *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition.

[5] Date, C. J. (2012). *Database Design and Relational Theory: Normal Forms and All That Jazz*. O'Reilly Media.

[6] Delis, A. and Tsotras, V. J. (2009). *Indexed Sequential Access Method*, pages 1435–1438. Springer US, Boston, MA.

[7] Dietrich, S. W. and Urban, S. D. (2011). *Fundamentals of Object Databases: Object Oriented and Object Relational Design*. Morgan, 1st edition.

[8] Du, W. and Elmagarmid, A. K. (1989). Quasi serializability: a correctness criterion for global concurrency control in interbase. In *Very Large Data Bases Conference*.

[9] Elmasri, R. and Navathe, S. (2010). *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition.

[10] Gligor, V. and Popescu-Zeletin, R. (1986). Transaction management in distributed heterogeneous database management systems. *Information Systems*, 11(4):287–297.

[11] Kirsten, W., Ihringer, M., Schulte, P., and Röhrig, B. (2001). *Object-Oriented Application Development Using the Caché Postrelational Database*. Springer Berlin Heidelberg.

[12] Maier, D. (1983). *Theory of Relational Databases*. Computer Science Press.

[13] Melton, J. and Buxton, S. (2006). *Querying XML : XQuery, XPath, and SQL/XML in context*. Margan Kaufmann, Amsterdam [etc.].

[14] Pratt, P. J. (2014). *Concepts of Database Management*. Cengage Learning, 8th edition.

[15] Rumbaugh, J. (2004). *The Unified Modeling Language Reference Manual (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.

[16] Singha, N. P. and Gupta, C. (2014). *Relational Database Management Systems*. Abhishek Publications.

[17] Strickland, R. (2016). *Cassandra 3.x High Availability - Second Edition*. Packt Publishing.

[18] Walmsley, P. (2007). *XQuery*. O'Reilly, 1st ed edition.