LECTURE NOTES
ON
DATABASE ENGINEERING

B. TECH II YEAR

By
Rashmi Ranjan Sahoo

DEPARTMENT OF COMPUTER SCIENCE AND ENGINGINEERING

PARALA MAHARAJA ENGINEERING COLLEGE

DATABASE MANAGEMENT SYSTEMS

Objectives:

- To Understand the basic concepts and the applications of database systems
- To Master the basics of SQL and construct queries using SQL
- To understand the relational database design principles
- To become familiar with the basic issues of transaction processing and concurrency control
- To become familiar with database storage structures and access techniques

UNIT I:

Database System Applications, Purpose of Database Systems, View of Data – Data Abstraction –Instances and Schemas – Data Models – the ER Model – Relational Model – Other Models – Database Languages – DDL – DML – database Access for applications Programs – Database Users and Administrator – Transaction Management – Database Architecture – Storage Manager – the Query Processor.

Introduction to the Relational Model – Structure – Database Schema, Keys – Schema Diagrams.

Database design and ER diagrams – ER Model - Entities, Attributes and Entity sets – Relationships and Relationship sets – ER Design Issues – Concept Design – Conceptual Design with relevant Examples. Relational Query Languages, Relational Operations.

UNIT II:

Relational Algebra – Selection and projection set operations – renaming – Joins – Division – Examples of Algebra overviews – Relational calculus – Tuple Relational Calculus (TRC) – Domain relational calculus (DRC).

Overview of the SQL Query Language – Basic Structure of SQL Queries, Set Operations, Aggregate Functions – GROUPBY – HAVING, Nested Sub queries, Views, Triggers, Procedures.

UNIT III:

Normalization – Introduction, Non loss decomposition and functional dependencies, First, Second, and third normal forms – dependency preservation, Boyce/Codd normal form.

Higher Normal Forms - Introduction, Multi-valued dependencies and Fourth normal form, Join dependencies and Fifth normal form

UNIT IV:

Transaction Concept- Transaction State- Implementation of Atomicity and Durability – Concurrent Executions – Serializability- Recoverability – Implementation of Isolation – Testing for serializability- Lock –Based Protocols – Timestamp Based Protocols- Validation-Based Protocols – Multiple Granularity.

UNIT V:

Recovery and Atomicity – Log – Based Recovery – Recovery with Concurrent Transactions – Check Points - Buffer Management – Failure with loss of nonvolatile storage-Advance Recovery systems- ARIES Algorithm, Remote Backup systems.

File organization – various kinds of indexes - B+ Trees- Query Processing – Relational Query Optimization.

TEXT BOOKS:

1. Database System Concepts, Silberschatz, Korth, McGraw hill, Sixth Edition.(All UNITS except III th)
2. Database Management Systems, Raghu Ramakrishnan, Johannes Gehrke, TATA McGrawHill 3rd Edition.

REFERENCE BOOKS:

1. Fundamentals of Database Systems, Elmasri Navathe Pearson Education.
2. An Introduction to Database systems, C.J. Date, A.Kannan, S.Swami Nadhan, Pearson, Eight Edition for UNIT III.

Outcomes:

- Demonstrate the basic elements of a relational database management system
- Ability to identify the data models for relevant problems
- Ability to design entity relationship and convert entity relationship diagrams into RDBMS and formulate SQL queries on the respect data
- Apply normalization for the development of application software's

INDEX

S. No	Unit	Topic
1	III	SCHEMA REFINEMENT
2	III	FUNCTIONAL DEPENDENCIES
3	III	NORMAL FORMS
4	III	DECOMPOSITIONS
5	III	DEPENDENCY-PRESERVING DECOMPOSITION INTO 3NF
6	III	OTHER KINDS OF DEPENDENCIES

S. No	Unit	Topic
1	IV	TRANSACTION CONCEPT
2	IV	CONCURRENT EXECUTION
3	IV	TRANSACTION CHARACTERISTICS
4	IV	RECOVERABLE SCHEDULES
5	IV	RECOVERY SYSTEM
6	IV	TIMESTAMP-BASED PROTOCOLS
7	IV	MULTIPLE GRANULARITY.

S. No	Unit	Topic
1	V	FAILURE WITH LOSS OF NON-VOLATILE STORAGE
2	V	REMOTE BACKUP
3	V	RECOVERY AND ATOMICITY
4	V	LOG-BASED RECOVERY
5	V	RECOVERY WITH CONCURRENT TRANSACTIONS
6	V	DBMS FILE STRUCTURE

MODULE-1

Introduction to Database Management System

As the name suggests, the database management system consists of two parts. They are:

1. Database and
2. Management System

What is a Database?

To find out what database is, we have to start from data, which is the basic building block of any DBMS.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).

Record: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

Roll	Name	Age
1	ABC	19

Table or **Relation**: Collection of related records.

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

The columns of this relation are called **Fields**, **Attributes** or **Domains**. The rows are called **Tuples** or **Records**.

Database: Collection of related relations. Consider the following collection of tables:

T1

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

T2

Roll	Address
1	KOL
2	DEL
3	MUM

T3 **T4**

Roll	Year
1	I
2	II
3	I

Year	Hostel
I	H1
II	H2

Age and ***Hostel*** attributes are in different tables.

A database in a DBMS could be viewed by lots of different people with different responsibilities.

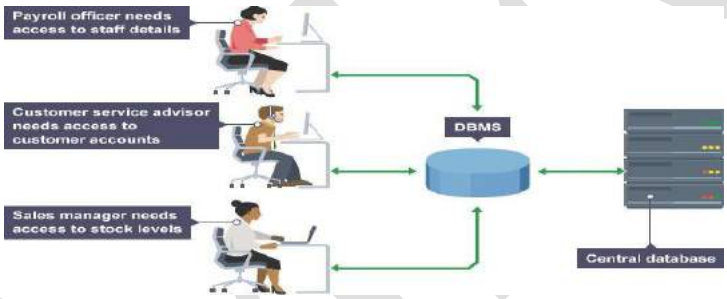


Figure 1.1: Empolyees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to see different kinds of data. Each employee in the company will have different levels of access to the database with their own customized **front-end** application.

In a database, data is organized strictly in row and column format. The rows are called **Tuple** or **Record**. The data items within one row may belong to different data types. On the other hand, the columns are often called **Domain** or **Attribute**. All the data items within a single attribute are of the same data type.

What is Management System?

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an

enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. By **data,** we mean known facts that can be recorded and that have implicit meaning.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Database Management System (DBMS) and Its Applications:

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

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

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

Enterprise Information

- ◦*Sales*: For customer, product, and purchase information.
- ◦*Accounting*: For payments, receipts, account balances, assets and other accounting information.
- ◦*Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
- ◦*Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.

Online retailers: For sales data noted above plus online order tracking, generation of recommendation lists, and
maintenance of online product evaluations.

- ◦*Banking*: For customer information, accounts, loans, and banking transactions.
- ◦*Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- ◦*Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and
  automated trading by the firm.
- •*Universities*: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- •*Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- •*Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

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

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems. Keeping organizational information in a file-processing system has a number of major disadvantages:

Data redundancy and inconsistency. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

Difficulty in accessing data. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* students.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure.

Consider a program to transfer $500 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the

$500 was removed from the balance of department *A* but was not credited to the balance of department *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur.

That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or
$9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision.
But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

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

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems.

Advantages of DBMS:

Controlling of Redundancy: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

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

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

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

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

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

Enforcements of Standards: With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

Data Independence: Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.

Reduced Application Development and Maintenance Time: DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.
2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.
3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.
4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction

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

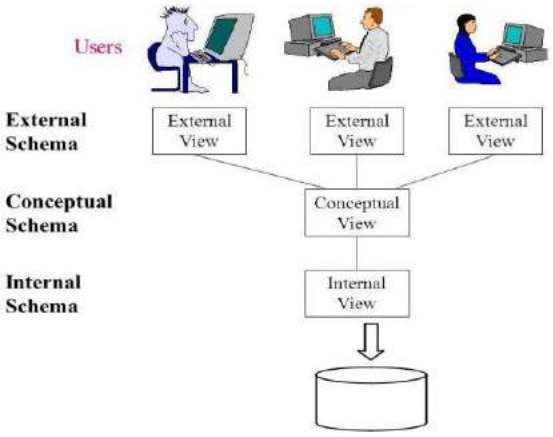


Figure 1.2 : Levels of Abstraction in a DBMS

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

•**Logical level (or Conceptual View / Schema)**: The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**.

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

For example, we may describe a record as follows:

type *instructor* = **record**
ID : **char** (5);
name : **char** (20);
dept name : **char** (20);
salary : **numeric** (8,2);

end;

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

- *department*, with fields *dept_name*, *building*, and *budget*
- *course*, with fields *course_id*, *title*, *dept_name*, and *credits*
- *student*, with fields *ID*, *name*, *dept_name*, and *tot_cred*

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive storage locations.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views.

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all. The concept of database schemas and instances can be understood by analogy to a program written in a programming language.

Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, which describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

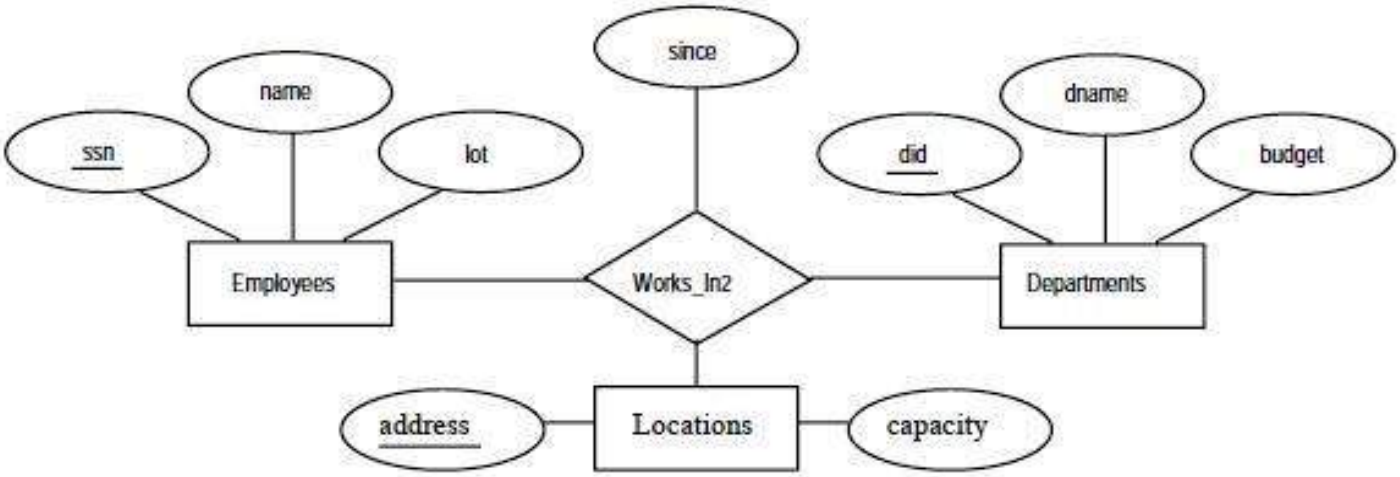
The data models can be classified into four different categories:

•**Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model.

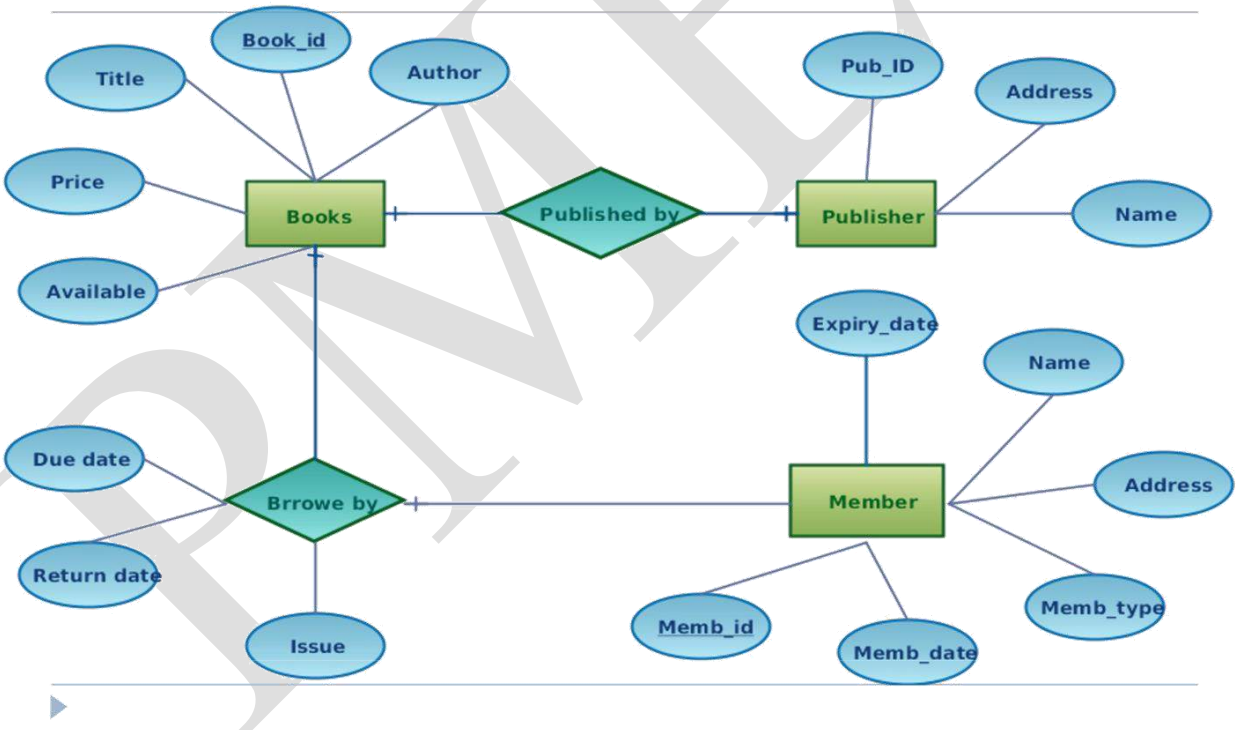
Entity-Relationship Model. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects.

Suppose that each department has offices in several locations and we want to record the locations at which each employee works. The ER diagram for this variant of Works In, which we call Works In2

Example - ternary



E-R Diagram for Library Management System



E R Model -(Banking Transaction System)

Object-Based Data Model. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object- oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity.

Semi-structured Data Model. The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semi-structured data.

Historically, the **network data model** and the **hierarchical data model** preceded the relational data model.
These models were tied closely to the underlying implementation, and complicated the task of modeling data.
As a result they are used little now, except in old database code that is still in service in some places.

Database Languages
A database system provides a **data-definition language** to specify the database schema and a **data-manipulation language** to express database queries and updates. In practice, the data-definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

Data-Manipulation Language

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

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

There are basically two types:
•**Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
•**Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

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

Data-Definition Language (DDL)
We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**). The DDL is also used to specify additional properties of the data.

•**Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

•**Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation.
•**Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion..

•**Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**,which contains **metadata**—that is, data about data.

Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database.

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.
Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program.
Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.
Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region).

Database Architecture:

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

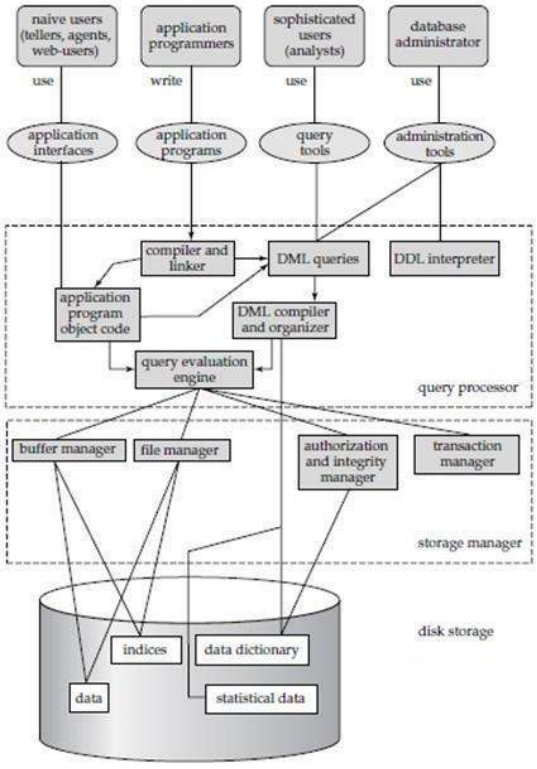


Figure 1.3: Database System Architecture

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor** components. The storage manager is important because databases

typically require a large amount of storage space. The query processor is important because it helps the database system simplify and facilitate access to data.

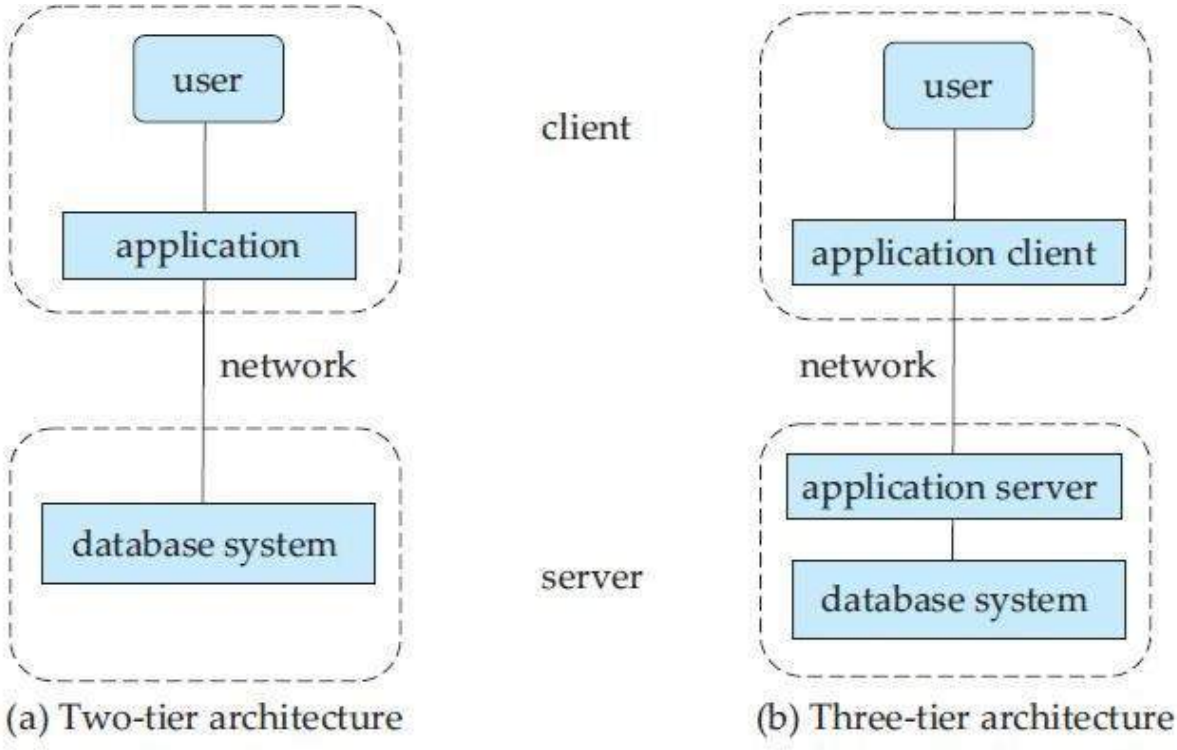


Figure 1.4: Two-tier and three-tier architectures.

Query Processor:

The query processor components include

· **DDL interpreter,** which interprets DDL statements and records the definitions in the data dictionary.

· **DML compiler,** which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

Storage Manager:

A *storage manager* is a program module that provides the interface between the lowlevel data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager.

The storage manager components include:

.

.

· **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

· **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.

· **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

· **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

Transaction Manager:

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints.

Conceptual Database Design - Entity Relationship(ER) Modeling:

Database Design Techniques

1. ER Modeling (Top down Approach)
2. Normalization (Bottom Up approach)

What is ER Modeling?

A graphical technique for understanding and organizing the data independent of the actual database implementation

We need to be familiar with the following terms to go further.

Entity

Any thing that has an independent existence and about which we collect data. It is also known as entity type.

In ER modeling, notation for entity is given below.

Entity instance

Entity instance is a particular member of the entity type.

Example for entity instance : A particular employee

Regular Entity

An entity which has its own key attribute is a regular entity.

Example for regular entity : Employee.

Weak entity

An entity which depends on other entity for its existence and doesn't have any key attribute of its own is a weak entity.

Example for a weak entity : In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity.

In ER modeling, notation for weak entity is given below.

Attributes

Properties/characteristics which describe entities are called attributes.

In ER modeling, notation for attribute is given below.

Domain of Attributes

The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set can be termed as the domain of the attribute day.

Key attribute

The attribute (or combination of attributes) which is unique for every entity instance is called key attribute.

E.g the employee_id of an employee, pan_card_number of a person etc.If the key attribute consists of two or more attributes in combination, it is called a composite key.

In ER modeling, notation for key attribute is given below.

Simple attribute

If an attribute cannot be divided into simpler components, it is a simple attribute.

Example for simple attribute : employee_id of an employee.

Composite attribute

If an attribute can be split into components, it is called a composite attribute.

Example for composite attribute : Name of the employee which can be split into First_name, Middle_name, and Last_name.

Single valued Attributes

If an attribute can take only a single value for each entity instance, it is a single valued attribute.

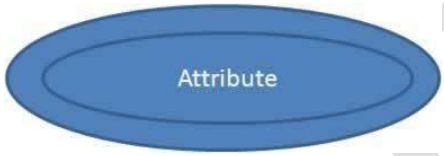
example for single valued attribute : age of a student. It can take only one value for a particular student.

Multi-valued Attributes

If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Multi-valued

example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers.

In ER modeling, notation for multi-valued attribute is given below.



Stored Attribute

An attribute which need to be stored permanently is a stored attribute

Example for stored attribute : name of a student

Derived Attribute

An attribute which can be calculated or derived based on other attributes is a derived attribute.

Example for derived attribute : age of employee which can be calculated from date of birth and current date.

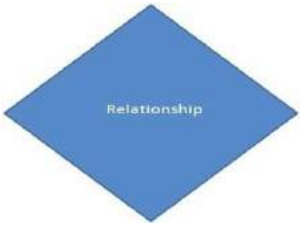
In ER modeling, notation for derived attribute is given below.

Relationships

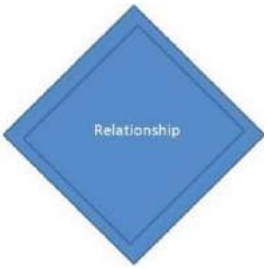
Associations between entities are called relationships

Example : An employee works for an organization. Here "works for" is a relation between the entities employee and organization.

In ER modeling, notation for relationship is given below.



However, in ER Modeling, to connect a weak Entity with others, you should use a weak relationship notation as given below



Degree of a Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary, where the degree is 1, 2, and 3, respectively.

Example for unary relationship: An employee is a manager of another employee Example for binary relationship: An employee works-for department. Example for ternary relationship: customer purchase item from a shop keeper **Cardinality of a Relationship**

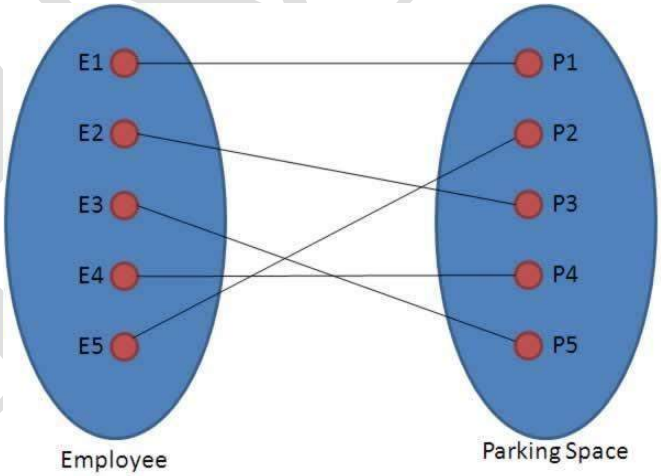
Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivity's as given below.

1. One to one (1:1) relationship
2. One to many (1:N) relationship
3. Many to one (M:1) relationship
4. Many to many (M:N) relationship

The minimum and maximum values of this connectivity is called the cardinality of the relationship

Example for Cardinality – One-to-One (1:1)

Employee is assigned with a parking space.



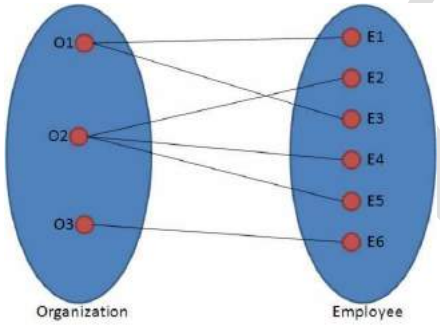
One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

In ER modeling, this can be mentioned using notations as given below

Example for Cardinality – One-to-Many (1:N)

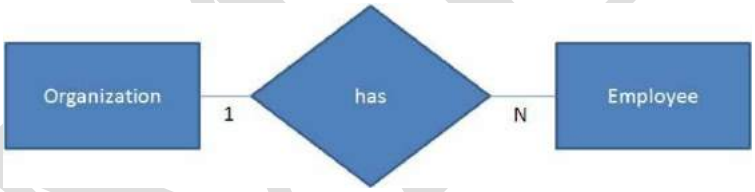
Organization has employees

One organization can have many employees , but one employee works in only one organization.



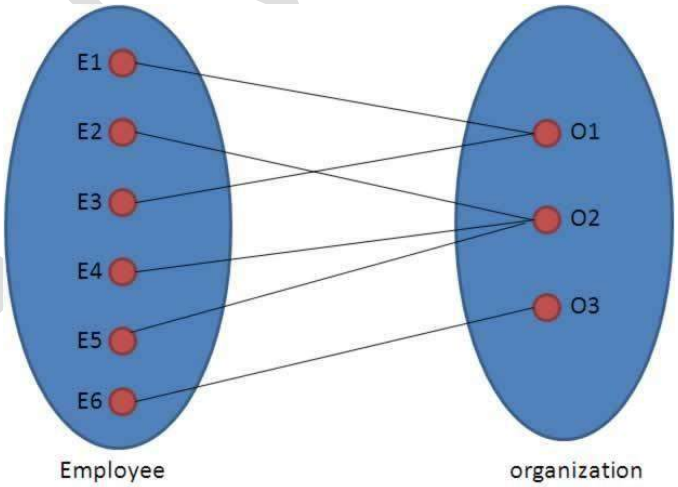
Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

In ER modeling, this can be mentioned using notations as given below



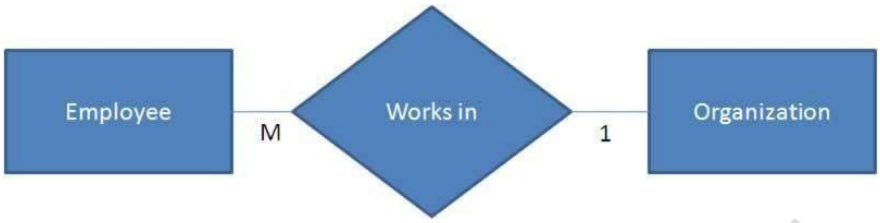
Example for Cardinality – Many-to-One (M :1)

It is the reverse of the One to Many relationship. employee works in organization



One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.

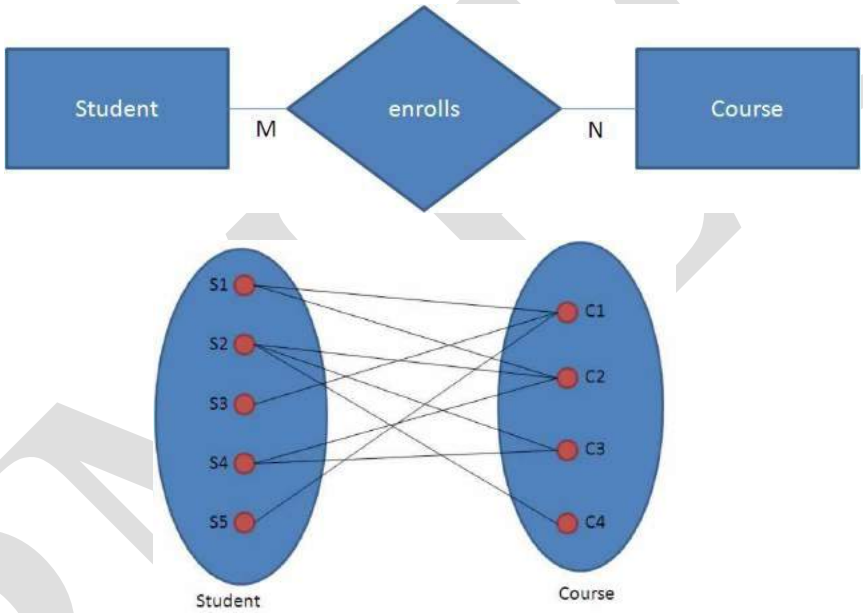


Cardinality – Many-to-Many (M:N)

Students enrolls for courses

One student can enroll for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)

In ER modeling, this can be mentioned using notations as given below



Relationship Participation

1. Total

In total participation, every entity instance will be connected through the relationship to another instance of the other participating entity types

2. Partial

Example for relationship participation

Consider the relationship - Employee is head of the department.

Here all employees will not be the head of the department. Only one employee will be the head of the department. In other words, only few instances of employee entity participate in the above relationship. So employee entity's participation is partial in the said relationship.

Advantages and Disadvantages of ER Modeling (Merits and Demerits of ER Modeling)

Advantages

1. ER Modeling is simple and easily understandable. It is represented in business users language and it can be understood by non-technical specialist.
2. Intuitive and helps in Physical Database creation.
3. Can be generalized and specialized based on needs.
4. Can help in database design.
5. Gives a higher level description of the system.

Disadvantages

1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.
2. Sometime diagrams may lead to misinterpretations

Relational Model

The relational model is today the primary data model for commercial data processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model.

Structure of Relational Databases:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure:1.5, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept name*, and *salary*.

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

Keys

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name.

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

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined. A relation, say $r1$, may include among its attributes the primary key of another relation, say $r2$. This attribute is called a **foreign key** from $r1$, referencing $r2$.

Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 1.12 shows the schema diagram for our university organization.

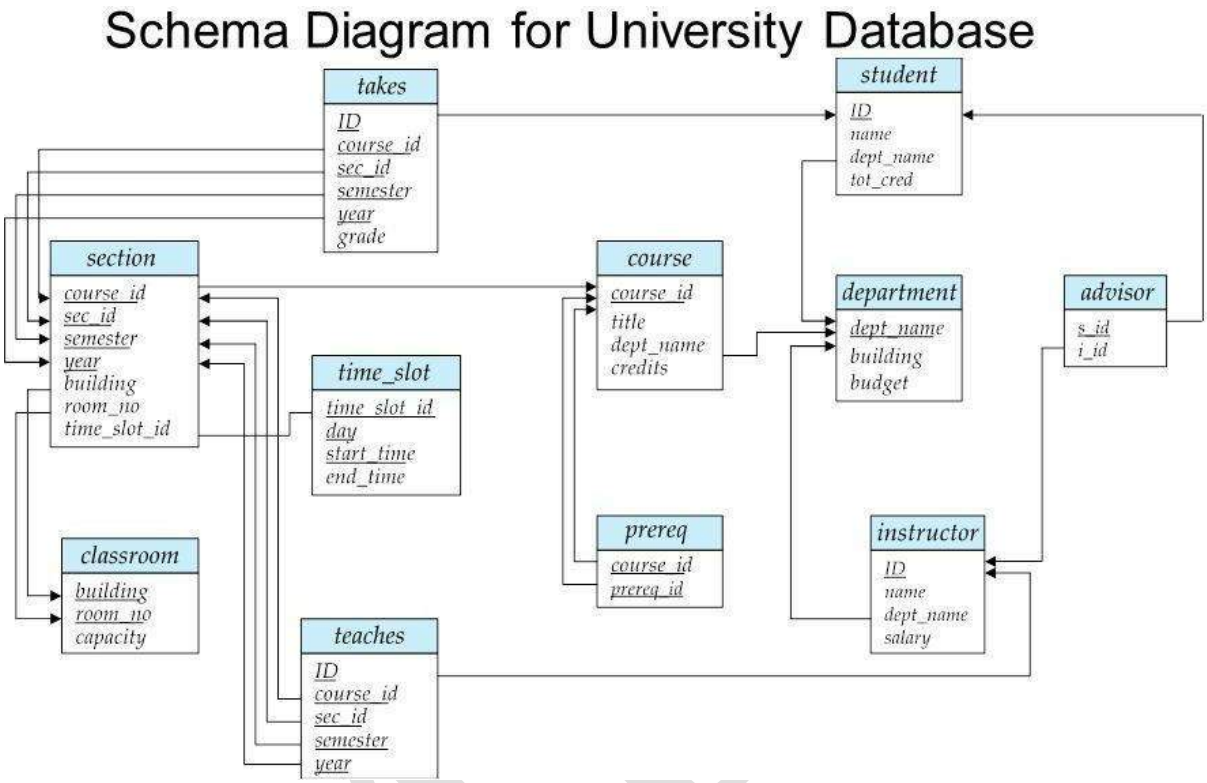


Figure 1.12: Schema diagram for the university database.

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram.

UNIT-2

Relational Algebra and Calculus

PRELIMINARIES

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances, and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances.

We present a number of sample queries using the following schema:

Sailors (*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
Boats (*bid:* integer, *bname:* string, *color:* string)
Reserves (*sid:* integer, *bid:* integer, *day:* date)

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations will be referred to by name, or positionally, using the order in which they are listed above.

RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the re- lational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) rela-tion instances as arguments and returns a relation instance as the result.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query.

Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation. Con - sider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression,

$$\sigma rating > 8(S2)$$

The selection operator σ specifies the tuples to retain through a *selection condition*. In general, the selection condition is a boolean combination (i.e., an expression using the logical connectives $\wedge$ and $\vee$) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <, <=, =, =, >=, or >.

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using π. The expression $\pi sname,rating(S2)$

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi age(S2)$$

a single tuple with *age=35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

Set Operations

The following standard operations on sets are also available in relational algebra: *union* (*U*), *intersection* ($\cap$), *set-difference* ($-$), and *cross-product* ($\times$).

- Union: $R \cup S$ returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *union-compatible*, and the schema of the result is defined to be identical to the schema of *R*.

- Intersection: $R \cap S$ returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

□Set-difference: $R - S$ returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

□Cross-product: $R \times S$ returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of $R \times S$ contains one tuple $\langle r, s \rangle$ (the concatenation of tuples r and s) for each pair of tuples $r \in R$, $s \in S$. The cross-product opertion is sometimes called Cartesian product.

sid	*sname*	*rating*	*age*
31	*Lubbe*	*8*	*55.5*
58	*Rusty*	*10*	*35.0*

Figure 4.9 $S1 \cap S2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

The result of the cross-product $S1 \times R1$ is shown in Figure 4.11 The fields in $S1 \times R1$ have the same domains as the corresponding fields in $R1$ and $S1$. In Figure 4.11 *sid* is listed in parentheses to

emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	*Dustin*	*7*	*45.0*	22	*101*	*10/10/96*
22	*Dustin*	*7*	*45.0*	58	*103*	*11/12/96*
31	*Lubber*	*8*	*55.5*	22	*101*	*10/10/96*
31	*Lubber*	*8*	*55.5*	58	*103*	*11/12/96*
58	*Rusty*	*10*	*35.0*	22	*101*	*10/10/96*
58	*Rusty*	*10*	*35.0*	58	*103*	*11/12/96*

Figure 4.11 $S1 \times R1$

Renaming

We introduce a renaming operator ρ for this purpose. The expression $\rho(R(F), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E, and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the *renaming list F*.

For example, the expression $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$ returns a relation that contains the tuples shown in Figure 4.11 and has the followi ng schema: C(*sid1:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer,*day:* dates).

Joins

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products.joins have received a lot of attention, and there are several variants of the join operation.

Condition Joins

The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

$$R \lhd\rhd c\ S = \sigma c(R \times S)$$

Thus $\lhd\rhd$ is defined to be a cross-product followed by a selection. Note that the condition *c* can (and typically *does*) refer to attributes of both *R* and *S*.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	*Dustin*	*7*	*45.0*	58	*103*	*11/12/96*
31	*Lubber*	*8*	*55.5*	58	*103*	*11/12/96*

Figure 4.12 $S1 \lhd\rhd S1.sid < R1.sid\ R1$

Equijoin

A common special case of the join operation $R \lhd\rhd S$ is when the *join condition* con-sists solely of equalities (connected by $\wedge$) of the form *R.name*1 = *S.name*2, that is, equalities between two fields in *R* and *S*. In this case, obviously, there is some redun-dancy in retaining both attributes in the result.

Natural Join

A further special case of the join operation $R \lhd\rhd S$ is an equijoin in which equalities are specified on *all* fields having the same name in *R* and *S*. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields.

Division

The division operator is useful for expressing certain kinds of queries, for example: "Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise.

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:
$\pi sname((\sigma bid=103 Reserves) \lhd\rhd Sailors)$

We first compute the set of tuples in Reserves with *bid* = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances *R*2 and *S*3, it yields a relation

(Q2) Find the names of sailors who have reserved a red boat.

$$\pi sname((\sigma color='red' Boats) \lhd\rhd Reserves \lhd\rhd Sailors$$

This query involves a series of two joins. First we choose (tuples describing) red boats.

(Q3) Find the colors of boats reserved by Lubber.

$$\pi color((\sigma sname='Lubber' Sailors) \lhd\rhd Reserves \lhd\rhd Boats)$$

This query is very similar to the query we used to compute sailors who reserved red boats. On instances *B*1, *R*2, and *S*3, the query will return the colors green and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi sname(Sailors \lhd\rhd Reserves)$$

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(T empboats, (\sigma color='red' Boats) \cup (\sigma color='green' Boats))$$
$$\pi sname(Tempboats \lhd\rhd Reserves \lhd\rhd Sailors)$$

(Q6) Find the names of sailors who have reserved a red and a green boat

$$\rho(T empboats2, (\sigma color='red' Boats) \cap (\sigma color='green' Boats))$$
$$\pi sname(Tempboats2 \lhd\rhd Reserves \lhd\rhd Sailors)$$

However, this solution is incorrect —it instead tries to compute sailors who have re-served a boat that is both red and green.

$$\rho(T empred, \pi sid((\sigma color='red' Boats) \lhd\rhd Reserves))$$
$$\rho(T empgreen, \pi sid((\sigma color='green' Boats) \lhd\rhd Reserves))$$
$$\pi sname((Tempred \cap Tempgreen) \lhd\rhd Sailors)$$

(Q7) Find the names of sailors who have reserved at least two boats.

$\rho(Reservations, \pi sid,sname,bid(Sailors \lhd\rhd Reserves))$

$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$

$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$

$\pi sname1\sigma(sid1=sid2) \cap (bid1=bid2)Reservationpairs$

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$\pi sid(\sigma age>20 Sailors) - \pi sid((\sigma color='red' Boats) \lhd\rhd Reserves \lhd\rhd Sailors)$

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors.

(Q9) Find the names of sailors who have reserved all boats.

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$\rho(T empsids, (\pi sid,bidReserves)/(\pi bidBoats))$
$\pi sname(Tempsids \lhd\rhd Sailors)$

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$\rho(T empsids, (\pi sid,bidReserves)/(\pi bid(\sigma bname='Interlake' Boats)))$
$\pi sname(Tempsids \lhd\rhd Sailors)$

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed.

Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields.

(Q11) Find all sailors with a rating above 7.

{S I S E *Sailors* ^ *S. rating* > 7*}*

with respect to the given database instance, *F* evaluates to (or simply 'is') true if one of the following holds:

- *F* is an atomic formula *R* ☐ *Rel*, and *R* is assigned a tuple in the instance of relation *Rel*.
- *F* is a comparison *R.a* op *S.b*, *R.a* op *constant*, or *constant* op *R.a*, and the tuples assigned to *R* and *S* have field values *R.a* and *S.b* that make the comparison true.
- *F* is of the form ¬*p*, and *p* is not true; or of the form *p* ^ *q*, and both *p* and *q* are true; or of the form *p V q*, and one of them is true, or of the form *p* ☐ *q* and *q* is true whenever[4] *p* is true.
- *F* is of the form ☐*R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*), including the variable *R*,[5] that makes the formula *p*(*R*) true.
- *F* is of the form ☐*R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*) that makes the formula *p*(*R*) true no matter what tuple is assigned to *R*.

(Q12) Find the names and ages of sailors with a rating above 7 .

{P | ☐S ☐ Sailors(*S.rating* > 7 ▯ *P.name* = *S.sname* ▯ *P.age* = *S.age*)*}*

This query illustrates a useful convention: *P* is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of *P* that are mentioned and *P* does not range over any of the relations in the query; that is, there is no subformula of the form *P* ☐ *Relname*.

(Q13) Find the sailor name, boat id, and reservation date for each reservation

{P | ☐*R* ☐*Reserves* ∃*S* ☐ *Sailors*
(*R.sid* = *S.sid* ▯ *P.bid* = *R.bid* ▯ *P.day* = *R.day* ▯ *P.sname* = *S.sname*)*}*

(Q1) Find the names of sailors who have reserved boat 103.

{P | ☐*S* ☐ *Sailors* ☐*R* ☐ *Reserves* (*R.sid* = *S.sid* ▯ *R.bid* = 103 ∧*P.sname* = *S.sname*)*}*

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the *sid* field, and with *bid* = 103."

(Q2) Find the names of sailors who have reserved a red boat.

$$\{P \mid \exists S \in Sailors\ \exists R \in Reserves\ (R.sid = S.sid \wedge P.sname = S.sname \wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that $S.sid = R.sid$, $R.bid = B.bid$, and $B.color = 'red'$."

(Q7) Find the names of sailors who have reserved at least two boats. $\{P \mid \exists S \in Sailors \wedge \exists R1 \in Reserves \wedge \exists R2 \in Reserves\ (S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$

(Q9) Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors \quad \forall B \in Boats \quad (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \wedge \forall B \in Boats \quad (B.color = 'red' \rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

Q9) Find the names of sailors who have reserved all boats.

$$\{ \langle N \rangle \mid \exists I, T, A(\langle I, N, T, A \rangle \in Sailors \wedge \forall B, BN, C(\neg(\langle B, BN, C \rangle \in Boats) \vee (\exists \langle Ir, Br, D \rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

THE FORM OF A BASIC SQL QUERY

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual evaluation strategy*. A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand, rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

Sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.1An Instance *S* 3 of SailorsFigure 5.2 An Instance *R*2 of Reserves

bid	*bname*	*color*
101	*Interlake*	*blue*
102	*Interlake*	*red*
103	*Clipper*	*green*
104	*Marine*	*red*

Q15) Find the names and ages of all sailors.

SELECT DISTINCT S.sname, S.age FROMSailors S

The answer to this query with and without the keyword DISTINCT on instance *S*3 of Sailors is shown in Figures 5.4 and 5.5. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35.

(Q11) Find all sailors with a rating above 7.

SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

(Q16) Find the sids of sailors who have reserved a red boat.

SELECTR.sid FROMBoats B, Reserves R WHERE B.bid = R.bid AND B.color = ‘red’

(Q2) Find the names of sailors who have reserved a red boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

(Q3) Find the colors of boats reserved by Lubber.

```
SELECT B.color FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid
AND R.bid = B.bid AND S.sname = 'Lubber'
```

(Q4) Find the names of sailors who have reserved at least one boat.

```
SELECT S.sname FROMSailors S, Reserves R WHERES.sid = R.sid
```

Expressions and Strings in the SELECT Command

SQL supports a more general version of the select-list than just a list of columns. Each item in a select-list can be of the form expression AS column name, where expression is any arithmetic or string expression over column names (possibly prefixed by range variables) and constants.

(Q5) Compute increments for the ratings of persons who have sailed two different boats on the same day.

```
SELECT S.sname, S.rating+1 AS rating FROM Sailors S, Reserves R1, Reserves R2
WHERE S.sid = R1.sid AND S.sid = R2.sid AND R1.day = R2.day AND R1.bid <>
R2.bid
```

Also, each item in a *qualification* can be as general as *expression1 = expression2*.

```
SELECT S1.sname AS name1, S2.sname AS name2 FROM Sailors S1, Sailors
S2 WHERE 2*S1.rating = S2.rating-1.
```

(Q6) Find the ages of sailors whose name begins and ends with B and has at least three characters.

```
SELECT S.age FROMSailors S WHERES.sname LIKE 'B %B'
```

The only such sailor is Bob, and his age is 63.5.

UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form pre-sented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT. [4] SQL also provides other set operations: IN (to check if an element is in a given set),op ANY,op ALL(tocom-pare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section. Consider the following query:

(Q1) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats
B2 WHERE S.sid = R1.sid AND R1.bid = B1.bid AND S.sid = R2.sid AND R2.bid
= B2.bid AND B1.color='red' AND B2.color = 'green'
```

(Q2) Find the* sids *of all sailors who have reserved red boats but not green boats.

```
SELECT S.sid FROMSailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' EXCEPT
SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2 WHERE
S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

NESTED QUERIES

A nested query is a querythat has another query embedded within it; the embedded query is called a subquery.

(Q1) Find the names of sailors who have reserved boat 103.

```
SELECT S.sname
FROMSailors S
WHERES.sid IN ( SELECT R.sid
                FROMReserves R
                WHERER.bid = 103 )
```

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT    S.sname
FROM    Sailors S
WHERE S.sid IN ( SELECT R.sid
                 FROMReserves R
                 WHERER.bid IN ( SELECT B.bid
                                 FROMBoats B
                                 WHEREB.color = 'red' )
```

(Q3) Find the names of sailors who have not ***reserved a red boat.***

```
SELECTS.sname
FROMSailors S
WHERE S.sid NOT IN ( SELECT R.sid
                     FROMReserves R
                     WHERER.bid IN ( SELECT B.bid
                                     FROMBoats B
                                     WHEREB.color = 'red' )
```

Correlated Nested Queries

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query:

(Q1) Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname
FROMSailors S
WHEREEXISTS ( SELECT *
              FROMReserves R
              WHERER.bid = 103
                    AND R.sid = S.sid )
```

Set-Comparison Operators

(Q1) Find sailors whose rating is better than some sailor called Horatio.

```
SELECT S.sid
FROMSailors S
WHERES.rating > ANY ( SELECT S2.rating
                      FROMSailors S2
                      WHERES2.sname = 'Horatio' )
```

(Q2) Find the sailors with the highest rating .

```
SELECT S.sid
FROM Sailors S
WHERE       S.rating >= ALL ( SELECT S2.rating
FROM Sailors S2 )
```

More Examples of Nested Queries

(Q1) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname
FROMSailors S, Reserves R, Boats B
WHERES.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
      AND S.sid IN ( SELECT S2.sid
                     FROMSailors S2, Boats B2, Reserves R2
                     WHERE S2.sid = R2.sid AND R2.bid = B2.bid
                           AND B2.color = 'green' )
```

Q9) Find the names of sailors who have reserved all boats.

```
SELECT S.sname
FROMSailors S
WHERENOT EXISTS (( SELECT B.bid
                   FROMBoats B )
                   EXCEPT
                   (SELECT R.bid
                   FROMReserves R
                   WHERER.sid = S.sid ))
```

AGGREGATE OPERATORS

We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM.

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

(Q1) Find the average age of all sailors.

```
SELECT AVG (S.age)
FROMSailors S
```

(Q2) Find the average age of sailors with a rating of 10.

```
SELECT AVG (S.age)
FROMSailors S
WHERES.rating = 10
```

```
SELECTS.sname,MAX(S.age)
FROMSailors S
```

Q3) Count the number of sailors.

```
SELECT COUNT (*)
FROMSailors S
```

The GROUP BY and HAVING Clauses

we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance).

(Q31) Find the age of the youngest sailor for each rating level.

```
SELECT MIN (S.age)
FROMSailors S
WHERES.rating = i
```

Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

```
SELECTS.rating, MIN (S.age) AS minageGROUP BY S.rating
HAVING   COUNT (*) > 1
```

More Examples of Aggregate Queries

Q3) For each red boat, find the number of reservations for this boat.

```
SELECT B.bid, COUNT (*) AS sailorcount FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red' GROUP BY B.bid
```

```
SELECT B.bid, COUNT (*) AS sailorcount FROM Boats B, Reserves R
WHERE R.bid = B.bid GROUP BY B.bid HAVING B.color = 'red'
```

(Q4) Find the average age of sailors for each rating level that has at least two sailors.

```
SELECTS.rating, AVG (S.age) AS avgage
FROMSailors S
GROUP BY S.rating
HAVING   COUNT (*) > 1
```

(Q5) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

```
SELECTS.rating, AVG ( S.age ) AS avgage
FROMSailors S
WHERE S. age >= 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)

                FROMSailors S2 WHERE S.rating = S2.rating
```

(Q6) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two such sailors.

```
SELECTS.rating, AVG ( S.age ) AS avgage
FROMSailors S
WHERE S. age > 18
GROUP BY S.rating

HAVING 1 < ( SELECT COUNT (*)
             FROMSailors S2
             WHERE S.rating = S2.rating AND S2.age >= 18 )
```

The above formulation of the query reflects the fact that it is a variant of Q35. The answer to Q36 on instance $S3$ is shown in Figure 5.16. It differs from the answer to Q35 in that there is no tuple for rating 10, since there is only one tuple with rating 10 and *age* ≥ 18.

```
SELECTS.rating, AVG ( S.age ) AS avgage
FROMSailors S
WHERE S. age > 18
GROUP BY S.rating
HAVING   COUNT (*) > 1
```

This formulation of Q36 takes advantage of the fact that the WHERE clause is applied before grouping is done; thus, only sailors with *age* > 18 are left when grouping is done. It is instructive to consider yet another way of writing this query:

```
SELECT Temp.rating, Temp.avgage
FROM ( SELECT S.rating, AVG ( S.age ) AS
                avgage, COUNT (*) AS
                ratingcount
       FROM Sailors S WHERE S. age > 18 GROUP BY S.rating ) AS Temp
WHERE Temp.ratingcount > 1
```

NULL VALUES

we have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a *rating* column, what row should we insert for Dan? What is needed here is a special value that denotes *unknown*.

SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable*. Using our Sailor table definition, we might enter the row $\langle 98, Dan, null, 39 \rangle$ to represent Dan. The presence of *null* values complicates many issues, and we consider the impact of *null* values on SQL in this section.

Comparisons Using Null Values

Consider a comparison such as *rating = 8*. If this is applied to the row for Dan, is this condition true or false? Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null*; for example, we can say *rating* IS NULL, which would evaluate to true on the row representing Dan. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for Dan.

Logical Connectives AND, OR, and NOT

Now, what about boolean expressions such as *rating* = 8 OR *age* < 40 and *rating* = 8 AND *age* < 40? Considering the row for Dan again, because *age* < 40, the first expression evaluates to true regardless of the value of *rating*, but what about the second? We can only say unknown.

INTRODUCTION TO VIEWS

A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view de nition. Consider the Students and Enrolled relations.

```
CREATE VIEW B-Students (name, sid, course)
        AS SELECT S.sname, S.sid, E.cid
            FROMStudents S, Enrolled E
            WHERES.sid = E.sid AND E.grade = `B'
```

This view can be used just like a base table, or explicitly stored table, in de ning new queries or views.

DESTROYING/ALTERING TABLES AND VIEWS

If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table de nition information), we can use the DROP TABLE command. For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any ref-erencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be speci ed. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

ALTER TABLE modi es the structure of an existing table. To add a column called *maiden-name* to Students, for example, we would use the following command:

```
ALTER TABLE Students
    ADD COLUMN maiden-name CHAR(10)
```

TRIGGERS

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

Event: A change to the database that activates the trigger.

Condition: A query or test that is run when the trigger is activated.

Action: A procedure that is executed when the trigger is activated and its con-dition is true.

A trigger *action* can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database.

Examples of Triggers in SQL

The examples shown in Figure 5.19, written using Oracle 7 Server syntax for defining triggers, illustrate the basic concepts behind triggers. (The SQL:1999 syntax for these triggers is similar; we will see an example using SQL:1999 syntax shortly.) The trigger called *init count* initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called *incr count* increments the counter for each inserted tuple that satisfies the condition *age* < 18.

```
CREATE TRIGGER init count BEFORE INSERT ON Students /* Event */
    DECLARE
        count INTEGER;
    BEGIN                                              /* Action */
        count := 0;
    END


CREATE TRIGGER incr count AFTER INSERT ON Students /* Event */
    WHEN (new. age < 18)      /* Condition; 'new' is just-inserted tuple */
    FOR EACH ROW
    BEGIN /* Action; a procedure in Oracle's PL/SQL syntax */ count :=
        count + 1;
    END
```

(identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with *age* < 18. (The trigger in Figure 5.19 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

```
CREATE TRIGGER set count AFTER INSERT ON Students /*       Event          */
REFERENCING NEW TABLE AS Inserted Tuples
FOR EACH STATEMENT
    INSERT/* Action */
        INTO Statistics Table (Modified Table,Modification Type,   Count)
        SELECT 'Students', 'Insert', COUNT *
        FROM Inserted Tuples I
        WHERE I.age < 18
```