DATABASE MANAGEMENT SYSTEM (21CS53)



**DATABASE MANAGEMENT SYSTEM [As per Choice Based Credit System (CBCS) [scheme] (Effective from the academic year 2022 -2023)**

**SEMESTER – V**

Subject Code

Number of Lecture Hours/Week

Total Number of Lecture Hours

**CREDITS – 03**

**Course objectives:** This course will enable students to

| 21CS53 |
| --- |
| 3 |
| 40 |

ν Provide a strong foundation in database concepts, technology, and practice. ν Practice SQL programming through a variety of database problems. ν Demonstrate the use of concurrency and transactions in database

ν Design and build database applications for real world problems.

**Module – 1 08 Hours**

**Introduction to Databases:** Introduction, Characteristics of database approach, Advantages of using the DBMS approach, History of database applications. **Overview of Database Languages and Architectures:** Data Models, Schemas, and Instances. Three schema architecture and data independence, database languages, and interfaces, The Database System environment. **Conceptual Data Modelling using Entities and Relationships:** Entity types, Entity sets, attributes, roles, and structural constraints, Weak entity types, ER diagrams, examples, Specialization and Generalization.

**Textbook 1:Ch 1.1 to 1.8, 2.1 to 2.6, 3.1 to 3.10**

**Module – 2 08 Hours Relational Model**: Relational Model Concepts, Relational Model Constraints and relational database schemas, Update operations, transactions, and dealing with constraint violations. **Relational Algebra:** Unary and Binary relational operations, additional relational operations (aggregate, grouping, etc.) Examples of Queries in relational algebra. **Mapping Conceptual Design into a Logical Design:** Relational Database Design using ER-to-Relational mapping. **SQL:** SQL data definition and data types, specifying constraints in SQL, retrieval queries in SQL, INSERT, DELETE, and UPDATE statements in SQL, Additional features of SQL.

**Textbook 1: Ch4.1 to 4.5, 5.1 to 5.3, 6.1 to 6.5, 8.1; Textbook 2: 3.5**

**Module – 3 08 Hours SQL : Advances Queries:** More complex SQL retrieval queries, Specifying constraints as assertions and action triggers, Views in SQL, Schema change statements in SQL. **Database Application Development:** Accessing databases from applications, An introduction to JDBC, JDBC classes and interfaces, SQLJ, Stored procedures, Case study: The internet Bookshop. **Internet Applications:** The three-Tier application architecture, The presentation layer, The Middle Tier **Textbook 1: Ch7.1 to 7.4; Tex book 2: 6.1 to 6.6, 7.5 to 7.7**

**Module – 4 08 Hours Normalization: Database Design Theory** – Introduction to Normalization using Functional and Multivalued Dependencies: Informal design guidelines for relation schema, Functional Dependencies, Normal Forms based on Primary

Keys, Second and Third Normal Forms, Boyce-Codd Normal Form, Multivalued Dependency and Fourth Normal Form, Join Dependencies and Fifth Normal Form. Normalization Algorithms: Inference Rules, Equivalence, and Minimal Cover, Properties of Relational Decompositions, Algorithms for Relational Database Schema Design, Nulls, Dangling tuples, and alternate Relational Designs, Further

discussion of Multivalued dependencies and 4NF, Other dependencies and Normal Forms

**Textbook 1: Ch14.1 to 14.7, 15.1 to 15.6**

**Module – 5 08 Hours Transaction Processing: Transaction Processing:** Introduction to Transaction Processing, Transaction and System concepts, Desirable properties of Transactions, Characterizing schedules based on recoverability, Characterizing schedules based on Serializability, Transaction support in SQL. **Concurrency Control in Databases:** Two-phase locking techniques for Concurrency control, Concurrency control based on Timestamp ordering, Multiversion Concurrency control techniques, Validation Concurrency control techniques, Granularity of Data items and Multiple Granularity Locking. **Introduction to Database Recovery Protocols:** Recovery Concepts, NO UNDO/REDO recovery based on Deferred update, Recovery techniques based on immediate update, Shadow paging, Database backup and recovery from catastrophic failures **Textbook 1: 20.1 to 20.6, 21.1 to 21.7, 22.1 to 22.4, 22.7.**

**Course outcomes:** The students should be able to:

1. Identify, analyze and define database objects, enforce integrity constraints on a database using RDBMS.

2. Use Structured Query Language (SQL) for database manipulation.

3. Design and build simple database systems

4. Develop application to interact with databases.

**Text Books:**

• Database systems Models, Languages, Design and Application Programming, Ramez Elmasri and Shamkant B. Navathe, 7th Edition, 2017, Pearson.

• .Database management systems, Ramakrishnan, and Gehrke, 3rd Edition, 2014, McGraw Hill

**Reference Books:**

• Silberschatz Korth and Sudharshan, Database System Concepts, 6th Edition, Mc GrawHill, 2013.

• Coronel, Morris, and Rob, Database Principles Fundamentals of Design, Implementation and Management, Cengage Learning 2012.

DATABASE MANAGEMENT SYSTEM (21CS53)



**INDEX SHEET**

**Contents Page No Module 1 Databases and Database Users** 2-19

1.1 Introduction

1.2: An Example:

1.3: Characteristics of the Database Approach:

1.4: Actors on the Scene

1.5: Workers Behind the Scene

1.6: Capabilities/Advantages of DBMS's

1.7: A Brief History of Database Applications

1.8: When Not to Use a DBMS

**Module 1ENTITY.1 Data Models, Schemas, and Instances** 20-35 2.1-RELATIONSHIP MODEL

2.1.2: Schemas, Instances, and Database State

2.2 DBMS Architecture and Data Independence

2.2.1: Three-Schema Architecture

2.3 Database Languages and Interfaces

2.3.1 DBMS Languages

2.3.2 DBMS Interfaces

2.4 Database System Environment

2.5 Centralized and Client/Server Architectures for DBMS's

2.6 Classification of DBMS's

2.7.Modeling Using the Entity-Relationship Model

Mr. MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 3

DATABASE MANAGEMENT SYSTEM (21CS53)



2.8 Entity-Relationship (ER) Model

2.8.1 Entities and Attributes

2.8.2: Entity Types, Entity Sets, Keys, and Domains

2.8.3 Initial Conceptual Design of COMPANY database

2.9 Relationship Types, Sets, Roles, and Structural Constraints

2.9.1:Ordering of entity types in relationship types

2.9.2 Degree of a relationship type

2.9.3 Constraints on Relationship Types

2.9.4 Attributes of Relationship Types

2.10 Weak Entity Types

**MODULE 2 The Relational Data Model and Relational Database Constraints** 36-57 **and Relational Algebra**

3.1 Relational Model Concepts

3.1.2 Characteristics of Relations

3.1.3 Relational Model Notation

3.2 Relational Model Constraints and Relational Database Schemas

3.2.1 Domain Constraints

3.2.2 Key Constraints

3.2.3 Relational Databases and Relational Database Schemas

3.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

3.3 Update Operations and Dealing with Constraint Violations

3.3.1 Insert

3.3.2 Delete

3.3.3 Update:

3.3.4 Transactions and dealing with constraints

Mr. MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 4

DATABASE MANAGEMENT SYSTEM (21CS53)



3.4 Relational Operation

3.5 Relational algebra operation Set theory Operations

3.6 JOIN Operations

3.7 Additional Relational Operations

3.8 Examples of Queries in Relational Algebra

3.9 Relational Database Design Using ER-to-Relational Mapping

**MODULE 2 SQL The Relational Database Standard** 58-69 4.1 Data Definition, Constraints, and Schema Changes in SQL2

4.2 Basic Queries in SQL

4.3 More Complex SQL Queries

**MODULE 3 SQL The Relational Database Standard** 70-80 5.1 Update Statements in SQL

5.2 Views in SQL

5.3 Additional features

5.4 Database Programming

5.5 Embedded SQL

5.6 Dynamic SQL

5.7 Database stored procedures and SQL/PSM

**MODULE 4Data Base design-1** 81-94 6.1 Informal design guidelines for relation schemas

6.1.1 Semantics of relations attributes

6.2. Inference Rules

6.3 Normalization

6.3.1 First Normal Form (1NF)

6.3.2 Second Normal Form (2NF)

MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 5

DATABASE MANAGEMENT SYSTEM (21CS53)



6.3.3 Third Normal Form (3NF

6.4 Boyce-Codd Normal Form (BCNF)

**MODULE 4 Data base design 2** 95-102 7.1 Properties of relational decomposition

7.2 Algorithms for Relational Database Schema Design

7.2.1 Decomposition and Dependency Preservation

7.2.2 Lossless-join Dependency

7.3 Multivolume Dependencies and Fourth Normal Form (4NF)

7.3.1 Fourth Normal Form (4NF)

7.4 Join Dependencies and 5 NF

7.5 Other dependencies:

7.5.1 Template Dependencies

7.5.2 Domain Key Normal Form

**UNIT 8 Transaction Processing Concepts** 103-110 8.1 Introduction to Transaction Processing

8.2 Transactions, Read and Write Operations

8.3 Why Concurrency Control Is Needed

8.4 Why Recovery Is Needed

8.5 Transaction and System Concepts

8.6 The System Log

8.7 Desirable Properties of Transactions

8.8 Schedules and Recoverability

8.10 Characterizing Schedules Based on

Assistant Professor Dept of CSE, Page 6

DATABASE MANAGEMENT SYSTEM (21CS53)

 **Module**-**1**

**Databases and Database Users**

Subject Code : **21CS53**

Exam Hours : 03

Databases and Database Users 1.1 Introduction

1.2: An Example:

IA Marks : 25

Total No. of Lecture Hours : 50

No. of Lecture Hours/Week : 04 Exam Marks : 80

1.3: Characteristics of the Database Approach:

1.4: Actors on the Scene

1.5: Workers Behind the Scene

1.6: Capabilities/Advantages of DBMS's

1.7: A Brief History of Database Applications

1.8: When Not to Use a DBMS

Mr. MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 8



DATABASE MANAGEMENT SYSTEM (21CS53)



**Module 1 Databases and Database Users**

**1.1 Introduction**

**Importance:** Database systems have become an essential component of life in modern society, in that many frequently occurring events trigger the accessing of at least one database: bibliographic library searches, bank transactions, hotel/airline reservations, grocery store purchases, online (Web) purchases, etc., etc.

**Traditional vs. more recent applications of databases**:

The applications mentioned above are all "traditional" ones for which the use of rigidly structured textual and numeric data suffices. Recent advances have led to the application of database technology to a wider class of data. Examples include **multimedia** databases (involving pictures, video clips, and sound messages) and **geographic** databases (involving maps, satellite images).

Also, database search techniques are applied by some WWW search engines. ***Definitions***

The term **database** is often used, rather loosely, to refer to just about any collection of related data. E&N say that, in addition to being a collection of related data, a database must have the following properties:

• It represents some aspect of the real (or an imagined) world, called the **miniworld** or **universe of discourse**. Changes to the miniworld are reflected in the database. Imagine, for example, a UNIVERSITY miniworld concerned with students, courses, course sections, grades, and course prerequisites.

• It is a logically coherent collection of data, to which some meaning can be attached. (Logical coherency requires, in part, that the database not be self-contradictory.)

• It has a purpose: there is an intended group of users and some preconceived applications that the users are interested in employing.

To summarize: a database has some source (i.e., the miniworld) from which data are derived, some degree of interaction with events in the represented miniworld (at least insofar as the data is updated when the state of the miniworld changes), and an audience that is interested in using it.

**An Aside: data vs. information vs. knowledge**: Data is the representation of "facts" or "observations" whereas information refers to the meaning thereof (according to some interpretation). Knowledge, on the other hand, refers to the ability to use information to achieve intended ends.

Page 9

DATABASE MANAGEMENT SYSTEM (21CS53)



**Computerized vs. manual:** Not surprisingly (this being a CS course), our concern will be with computerized database systems, as opposed to manual ones, such as the card catalog-based systems that were used in libraries in ancient times (i.e., before the year 2000). (Some authors wouldn't even recognize a non-computerized collection of data as a database, but E&N do.)

**Size/Complexity:** Databases run the range from being small/simple (e.g., one person's recipe database) to being huge/complex (e.g., Amazon's database that keeps track of all its products, customers, and suppliers).

**Definition:** A **database management system** (DBMS) is a collection of programs enabling users to create and maintain a database.

More specifically, a DBMS is a *general purpose* software system facilitating each of the following (with respect to a database):

• **definition:** specifying data types (and other constraints to which the data must conform) and data organization

• **construction:** the process of storing the data on some medium (e.g., magnetic disk) that is controlled by the DBMS

• **manipulation:** querying, updating, report generation

• **sharing:** allowing multiple users and programs to access the database "simultaneously" • **system protection:** preventing database from becoming corrupted when hardware or software failures occur

• **security protection:** preventing unauthorized or malicious access to database.

Given all its responsibilities, it is not surprising that a typical DBMS is a complex piece of software.

A database together with the DBMS software is referred to as a **database system**. (See Figure 1.1, page 7.)

**1.2: An Example:**

UNIVERSITY database in Figure 1.2. Notice that it is relational!

Among the main ideas illustrated in this example is that each file/relation/table has a set of named fields/attributes/columns, each of which is specified to be of some data type. (In addition to a data type, we might put further restrictions upon a field, e.g., GRADE\_REPORT must have a value from the set {'A', 'B', ..., 'F'}.)

The idea is that, of course, each table will be populated with data in the form of records/tuples/rows, each of which represents some entity (in the miniworld) or some relationship between entities.

Page 10

DATABASE MANAGEMENT SYSTEM (21CS53)



For example, each record in the **STUDENT** table represents a —surprise!— student. Similarly for the **COURSE** and **SECTION** tables.

On the other hand, each record in **GRADE\_REPORT** represents a relationship between a student and a section of a course. And each record in **PREREQUISITE** represents a relationship between two courses.

Database *manipulation* involves *querying* and *updating*.

Examples of (informal) queries:

• Retrieve the transcript(s) of student(s) named 'Smith'.

• List the names of students who were enrolled in a section of the 'Database' course in Spring 2006, as well as their grades in that course section.

• List all prerequisites of the 'Database' course.

Examples of (informal) updates:

• Change the CLASS value of 'Smith' to sophomore (i.e., 2).

• Insert a record for a section of 'File Processing' for this semester.

• Remove from the prerequisites of course 'CMPS 340' the course 'CMPS 144'.

Of course, a query/update must be conveyed to the DBMS in a precise way (via the query language of the DBMS) in order to be processed.

As with software in general, developing a new database (or a new application for an existing database) proceeds in phases, including **requirements analysis** and various levels of **design** (conceptual (e.g., Entity-Relationship Modeling), logical (e.g., relational), and physical (file structures)).

**1.3: Characteristics of the Database Approach:**

**Database approach vs. File Processing approach:** Consider an organization/enterprise that is organized as a collection of departments/offices. Each department has certain data processing "needs", many of which are unique to it. In the **file processing approach**, each department would control a collection of relevant data files and software applications to manipulate that data.

For example, a university's Registrar's Office would maintain data (and programs) relevant to student grades and course enrollments. The Bursar's Office would maintain data (and programs) pertaining to fees owed by students for tuition, room and board, etc. (Most likely, the people in these offices would not be in direct possession of their data and programs, but rather the university's Information Technology Department would be responsible for providing services such as data storage, report generation, and programming.)



 Page 11

DATABASE MANAGEMENT SYSTEM (21CS53)



One result of this approach is, typically, **data redundancy**, which not only wastes storage space but also makes it more difficult to keep changing data items consistent with one another, as a change to one copy of a data item must be made to all of them (called **duplication-of-effort**). **Inconsistency** results when one (or more) copies of a datum are changed but not others. (E.g., If you change your address, informing the Registrar's Office should suffice to ensure that your grades are sent to the right place, but does not guarantee that your next bill will be, as the copy of your address "owned" by the Bursar's Office might not have been changed.)

In the **database approach**, a single repository of data is maintained that is used by all the departments in the organization. (Note that "single repository" is used in the logical sense. In physical terms, the data may be *distributed* among various sites, and possibly *mirrored*.)

**Main Characteristics of database approach:**

1. **Self-Description:** A database system includes —in addition to the data stored that is of relevance to the organization— a complete definition/description of the database's structure and constraints. This **meta-data** (i.e., data about data) is stored in the so-called **system catalog**, which contains a description of the structure of each file, the type and storage format of each field, and the various constraints on the data (i.e., conditions that the data must satisfy).

See Figures 1.1 and 1.3.

The system catalog is used not only by users (e.g., who need to know the names of tables and attributes, and sometimes data type information and other things), but also by the DBMS software, which certainly needs to "know" how the data is structured/organized in order to interpret it in a manner consistent with that structure. Recall that a DBMS is *general purpose*, as opposed to being a specific database application. Hence, the structure

of the data cannot be "hard-coded" in its programs (such as is the case in typical *file processing* approaches), but rather must be treated as a "parameter" in some sense.

2. **Insulation between Programs and Data; Data Abstraction:**

**Program-Data Independence**: In traditional file processing, the structure of the data files accessed by an application is "hard-coded" in its source code. (E.g., Consider a file descriptor in a COBOL program: it gives a detailed description of the layout of the records in a file by describing, for each field, how many bytes it occupies.)

If, for some reason, we decide to change the structure of the data (e.g., by adding the first two digits to the YEAR field, in order to make the program Y2K compliant!), **every** application in which a description of that file's structure is hard-coded must be changed!

In contrast, DBMS access programs, in most cases, do not require such changes, because the structure of the data is described (in the system catalog) separately from the programs that access it and those programs consult the catalog in order to ascertain the structure of



Page 12

DATABASE MANAGEMENT SYSTEM (21CS53)



the data (i.e., providing a means by which to determine boundaries between records and between fields within records) so that they interpret that data properly.

See Figure 1.4.

In other words, the DBMS provides a conceptual or logical view of the data to application programs, so that the underlying implementation may be changed without the programs being modified. (This is referred to as *program-data independence*.)

Also, which access paths (e.g., indexes) exist are listed in the catalog, helping the DBMS to determine the most efficient way to search for items in response to a q uery.

**Data Abstraction:**

◼A **data model** is used to hide storage details and present the users with a conceptual view  of the database.

◼Programs refer to the data model constructs rather than data storage details

*Note:* In fairness to COBOL, it should be pointed out that it has a COPY feature that allows different application programs to make use of the same file descriptor stored in a "library". This provides some degree of program-data independence, but not nearly as much as a good DBMS does. *End of note.*

**Example** by which to illustrate this concept: Suppose that you are given the task of developing a program that displays the contents of a particular data file. Specifically, each record should be displayed as follows:

Record #i:

value of first field

value of second field

...

...

value of last field

To keep things very simple, suppose that the file in question has fixed-length records of 57 bytes with six fixed-length fields of lengths 12, 4, 17, 2, 15, and 7 bytes, respectively, all of which are ASCII strings. Developing such a program would not be difficult. However, the obvious solution would be tailored specifically for a file having the particular structure described here and would be of no use for a file with a different structure.

Now suppose that the problem is generalized to say that the program you are to develop must be able to display *any* file having fixed-length records with fixed-length fields that are ASCII strings. Impossible, you say? Well, yes, unless the program has the ability to access a description of the file's structure (i.e., lengths of its records and the fields



Page 13

DATABASE MANAGEMENT SYSTEM (21CS53)



therein), in which case the problem is not hard at all. This illustrates the power of metadata, i.e., data describing other data.

3. **Multiple Views of Data:** Different users (e.g., in different departments of an organization) have different "views" or perspectives on the database. For example, from the point of view of a Bursar's Office employee, student data does not include anything about which courses were taken or which grades were earned. (This is an example of a **subset** view.)

As another example, a Registrar's Office employee might think that GPA is a field of data in each student's record. In reality, the underlying database might calculate that value each time it is needed. This is called **virtual** (or **derived**) data.

A view designed for an academic advisor might give the appearance that the data is structured to point out the prerequisites of each course.

(See Figure 1.5, page 14.)

A good DBMS has facilities for defining multiple views. This is not only convenient for users, but also addresses security issues of data access. (E.g., The Registrar's Office view should not provide any means to access financial data.)

4. **Data Sharing and Multi-user Transaction Processing:** As you learned about (or will) in the OS course, the simultaneous access of computer resources by multiple users/processes is a major source of complexity. The same is true for multi-user DBMS's.

Arising from this is the need for **concurrency control**, which is supposed to ensure that several users trying to update the same data do so in a "controlled" manner so that the results of the updates are as though they were done in some sequential order (rather than interleaved, which could result in data being incorrect).

This gives rise to the concept of a **transaction**, which is a process that makes one or more accesses to a database and which must have the appearance of executing in *isolation* from all other transactions (even ones that access the same data at the "same time") and of being *atomic* (in the sense that, if the system crashes in the middle of its execution, the database contents must be as though it did not execute at all).

Applications such as **airline reservation systems** are known as **online transaction processing** applications.

**1.4: Actors on the Scene**

These apply to "large" databases, not "personal" databases that are defined, constructed, and used by a single person via, say, Microsoft Access.



 Page 14

DATABASE MANAGEMENT SYSTEM (21CS53)



◼Users may be divided into

◼Those who actually use and control the database content, and those who design, develop and  maintain database applications (called ―Actors on the Scene‖), and

◼Those who design and develop the DBMS software and related tools, and the computer  systems operators (called ―Workers Behind the Scene‖).

5. **Database Administrator** (DBA): This is the chief administrator, who oversees and manages the database system (including the data and software). Duties include authorizing users to access the database, coordinating/monitoring its use, acquiring hardware/software for upgrades, etc. In large organizations, the DBA might have a support staff.

6. **Database Designers**: They are responsible for identifying the data to be stored and for choosing an appropriate way to organize it. They also define **views** for different categories of users. The final design must be able to support the requirements of all the user sub-groups.

7. **End Users**: These are persons who access the database for **querying**, **updating**, and **report generation**. They are main reason for database's existence!

o **Casual end users:** use database occasionally, needing different information each time; use query language to specify their requests; typically middle- or high-level managers.

o **Naive/Parametric end users**: Typically the biggest group of users; frequently query/update the database using standard **canned transactions** that have been carefully programmed and tested in advance. Examples:

▪ bank tellers check account balances, post withdrawals/deposits

▪ reservation clerks for airlines, hotels, etc., check availability of

seats/rooms and make reservations.

▪ shipping clerks (e.g., at UPS) who use buttons, bar code scanners, etc., to update status of in-transit packages.

o **Sophisticated end users**: engineers, scientists, business analysts who implement their own applications to meet their complex needs.

• **Stand-alone users**: Use "personal" databases, possibly employing a special purpose (e.g., financial) software package. Mostly maintain personal databases using ready-to-use packaged applications.

o An example is a tax program user that creates its own internal database. o Another example is maintaining an address book

4. **System Analysts, Application Programmers, Software Engineers**:

o **System Analysts**: determine needs of end users, especially naive and parametric users, and develop specifications for canned transactions that meet these needs. o **Application Programmers**: Implement, test, document, and maintain programs that satisfy the specifications mentioned above.

**1.5: Workers Behind the Scene**

****

**** Page 15

DATABASE MANAGEMENT SYSTEM (21CS53)



• **DBMS system designers/implementors:** provide the DBMS software that is at the foundation of all this!

• **tool developers**: design and implement software tools facilitating database system design, performance monitoring, creation of graphical user interfaces, prototyping, ets. • **operators and maintenance personnel**: responsible for the day-to-day operation of the system.

**1.6: Capabilities/Advantages of DBMS's**

1. **Controlling Redundancy:** Data redundancy (such as tends to occur in the "file processing" approach) leads to **wasted storage space**, **duplication of effort** (when multiple copies of a datum need to be updated), and a higher liklihood of the introduction of **inconsistency**.

On the other hand, redundancy can be used to improve performance of queries. Indexes, for example, are entirely redundant, but help the DBMS in processing queries more quickly.

Another example of using redundancy to improve performance is to store an "extra" field in order to avoid the need to access other tables (as when doing a JOIN, for example). See Figure 1.6 (page 18): the StudentName and CourseNumber fields need not be there.

A DBMS should provide the capability to automatically enforce the rule that no inconsistencies are introduced when data is updated. (Figure 1.6 again, in which Student\_name does not match Student\_number.)

2. **Restricting Unauthorized Access:** A DBMS should provide a **security and authorization subsystem**, which is used for specifying restrictions on user accounts. Common kinds of restrictions are to allow read-only access (no updating), or access only to a subset of the data (e.g., recall the Bursar's and Registrar's office examples from above).

3. **Providing Persistent Storage for Program Objects:** Object-oriented database systems make it easier for complex runtime objects (e.g., lists, trees) to be saved in secondary storage so as to survive beyond program termination and to be retrievable at a later time.

4. **Providing Storage Structures for Efficient Query Processing:** The DBMS maintains indexes (typically in the form of trees and/or hash tables) that are utilized to improve the execution time of queries and updates. (The choice of which indexes to create and maintain is part of *physical database design and tuning* (see Chapter 16) and is the responsibility of the DBA.

The **query processing and optimization** module is responsible for choosing an efficient query execution plan for each query submitted to the system. (See Chapter 15.)

5. **Providing Backup and Recovery:** The subsystem having this responsibility ensures that recovery is possible in the case of a system crash during execution of one or more transactions.



 Page 16

DATABASE MANAGEMENT SYSTEM (21CS53)



6. **Providing Multiple User Interfaces:** For example, query languages for casual users, programming language interfaces for application programmers, forms and/or command codes for parametric users, menu-driven interfaces for stand-alone users.

7. **Representing Complex Relationships Among Data:** A DBMS should have the capability to represent such relationships and to retrieve related data quickly.

8. **Enforcing Integrity Constraints:** Most database applications are such that the semantics (i.e., meaning) of the data require that it satisfy certain restrictions in order to make sense. Perhaps the most fundamental constraint on a data item is its data type, which specifies the universe of values from which its value may be drawn. (E.g., a Grade field could be defined to be of type Grade\_Type, which, say, we have defined as including precisely the values in the set { "A", "A-", "B+", ..., "F" }.

Another kind of constraint is *referential integrity*, which says that if the database includes an entity that refers to another one, the latter entity must exist in the database. For example, if (R56547, CIL102) is a tuple in the Enrolled\_In relation, indicating that a student with ID R56547 is taking a course with ID CIL102, there *must be* a tuple in the Student

relation corresponding to a student with that ID.

9. **Permitting Inferencing and Actions Via Rules:** In a **deductive** database system, one may specify *declarative* rules that allow the database to infer new data! E.g., Figure out which students are on academic probation. Such capabilities would take the place of application programs that would be used to ascertain such information otherwise.

**Active** database systems go one step further by allowing "active rules" that can be used to initiate actions automatically.

**1.7: A Brief History of Database Applications**

◼Early Database Applications:

◼The Hierarchical and Network Models were introduced in mid 1960s and dominated during  the seventies.

◼A bulk of the worldwide database processing still occurs using these models.

◼Relational Model based Systems:

◼Relational model was originally introduced in 1970, was heavily researched and  experimented with in IBM Research and several universities.

◼Object-oriented and emerging applications:

Object-Oriented Database Management Systems (OODBMSs) were introduced in late 1980s and early 1990s to cater to the need of complex data processing in CAD and other applications. ◼Their use has not taken off much.

Many relational DBMSs have incorporated object database concepts, leading to a new category called *object-relationa*l DBMSs (ORDBMSs)



 Page 17

DATABASE MANAGEMENT SYSTEM (21CS53)



*Extended relational* systems add further capabilities (e.g. for multimedia data, XML, and other data types)

◼Relational DBMS Products emerged in the 1980s

◼Data on the Web and E-commerce Applications:

◼Web contains data in HTML (Hypertext markup language) with links among pages. ◼This has given rise to a new set of applications and E-commerce is using new standards  like XML (eXtended Markup Language).

◼Script programming languages such as PHP and JavaScript allow generation of dynamic  Web pages that are partially generated from a database

◼New functionality is being added to DBMSs in the following areas:

◼Scientific Applications

◼XML (eXtensible Markup Language)

◼Image Storage and Management

◼Audio and Video data management

◼Data Warehousing and Data Mining

◼Spatial data management

◼Time Series and Historical Data Management

◼The above gives rise to *new research and development* in incorporating  new data types, complex data structures, new operations and storage and indexing schemes in database systems.

◼Also allow database updates through Web pages

**1.8: When Not to Use a DBMS**

**Main inhibitors (costs) of using a DBMS:**

◼High initial investment and possible need for additional hardware.

◼Overhead for providing generality, security, concurrency control, recovery, and integrity  functions.

◼When a DBMS may be unnecessary:

◼If the database and applications are simple, well defined, and not expected to change. ◼If there are stringent real-time requirements that may not be met because of DBMS  overhead.

◼If access to data by multiple users is not required.

◼When no DBMS may suffice:

◼If the database system is not able to handle the complexity of data because of modeling  limitations

◼If the database users need special operations not supported by the DBMS.



 Page 18

DATABASE MANAGEMENT SYSTEM (21CS53)



Questions

1. Define the following terms:

i) Data base

ii) Canned transaction

iii) Data model:

iv) Meta data:

v) Data base designer

2. Explain the characteristics of data base approach.

3. Discuss the main Characteristics of the database approach.How does it differ from Traditional file systems?

4. Explain the difference between Logical and physical data Independence? 5. A.Breifly discuss the advantages of using the DBMS

6. Explain the componenet modules of DBMS and their interaction,with help of a diagram 7. Explain the typical components of a DBMS with a neat diagram.

8. Define and explain the following terms with an example for each.

9. What is meant by ―persistent storage for program objects‖. Explain.



 Page 19

DATABASE MANAGEMENT SYSTEM (21CS53)

**ENTITY.1 Data Models, Schemas, and Instances**

Subject Code : **21CS53** Exam Hours : 03

IA Marks : 25

Total No. of Lecture Hours : 50

No. of Lecture Hours/Week : 04 Exam Marks : 80

**ENTITY.1 Data Models, Schemas, and Instances**

2.1-RELATIONSHIP MODEL

2.1.2: Schemas, Instances, and Database State

2.2 DBMS Architecture and Data Independence

2.2.1: Three-Schema Architecture

2.3 Database Languages and Interfaces

2.3.1 DBMS Languages

2.3.2 DBMS Interfaces

2.4 Database System Environment

2.5 Centralized and Client/Server Architectures for DBMS's

2.6 Classification of DBMS's

2.7.Modeling Using the Entity-Relationship Model

2.8 Entity-Relationship (ER) Model

2.8.1 Entities and Attributes

2.8.2: Entity Types, Entity Sets, Keys, and Domains

2.8.3 Initial Conceptual Design of COMPANY database

2.9 Relationship Types, Sets, Roles, and Structural Constraints

2.9.1:Ordering of entity types in relationship types

2.9.2 Degree of a relationship type

2.9.3 Constraints on Relationship Types



 Page 20

DATABASE MANAGEMENT SYSTEM (21CS53)



2.9.4 Attributes of Relationship Types

2.10 Weak Entity Types



Page 21

DATABASE MANAGEMENT SYSTEM (21CS53)



**ENTITY-RELATIONSHIP MODEL**

**2.1 Data Models, Schemas, and Instances**

One fundamental characteristic of the database approach is that it provides some level of *data abstraction* by hiding details of data storage that are irrelevant to database users.

A **data model** ---a collection of concepts that can be used to describe the conceptual/logical structure of a database--- provides the necessary means to achieve this abstraction.

By *structure* is meant the data types, relationships, and constraints that should hold for the data. Most data models also include a set of **basic operations** for specifying retrievals/updates.

Object-oriented data models include the idea of objects having behavior (i.e., applicable methods) being stored in the database (as opposed to purely "passive" data).

According to C.J. Date (one of the leading database experts), a **data model** is an abstract, self contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users interact. The objects allow us to model the *structure* of data; the operators allow us to model its *behavior*.

In the *relational* data model, data is viewed as being organized in two-dimensional tables comprised of tuples of attribute values. This model has operations such as Project, Select, and Join.

A data model is not to be confused with its **implementation**, which is a physical realization on a real machine of the components of the *abstract machine* that together constitute that model.

Logical vs. physical!!

There are other well-known data models that have been the basis for database systems. The best known models pre-dating the relational model are the **hierarchical** (in which the entity types form a tree) and the **network** (in which the entity types and relationships between them form a graph).

**Categories of Data Models** (based on degree of abstractness):

• **high-level/conceptual**: (e.g., ER model of Chapter 3) provides a view close to the way users would perceive data; uses concepts such as

o **entity**: real-world object or concept (e.g., student, employee, course, department, event)

o **attribute**: some property of interest describing an entity (e.g., height, age, color) o **relationship**: an interaction among entities (e.g., works-on relationship between an employee and a project)



 Page 22

DATABASE MANAGEMENT SYSTEM (21CS53)



• **representational/implementational**: intermediate level of abstractness; example is relational data model (or the network model alluded to earlier). Also called **record-based** model.

• **low-level/physical**: gives details as to how data is stored in computer system, such as record formats, orderings of records, access paths (indexes). (See Chapters 13-14.)

**2.1.2: Schemas, Instances, and Database State**

One must distinguish between the *description* of a database and the database itself. The former is called the **database schema**, which is specified during design and is not expected to change often. (See Figure 2.1, p. 33, for schema diagram for relational UNIVERSITY database.)

The actual data stored in the database probably changes often. The data in the database at a particular time is called the **state** of the database, or a **snapshot**.

Application requirements change occasionally, which is one of the reasons why software maintenance is important. On such occasions, a change to a database's schema may be called for. An example would be to add a Date\_of\_Birth field/attribute to the STUDENT table. Making changes to a database schema is known as **schema evolution**. Most modern DBMS's support schema evolution operations that can be applied while a database is operational.

**2.2 DBMS Architecture and Data Independence**

**2.2.1: Three-Schema Architecture**: (See Figure 2.2, page 34.) This idea was first described by the ANSI/SPARC committee in late 1970's. The goal is to separate (i.e., insert layers of "insulation" between) user applications and the physical database. C.J. Date points out that it is an ideal that few, if any, real-life DBMS's achieve fully.

• **internal level**: has an internal/physical schema that describes the physical storage structure of the database using a low-level data model)

• **conceptual level**: has a conceptual schema describing the (logical) structure of the whole database for a community of users. It hides physical storage details, concentrating upon describing entities, data types, relationships, user operations, and constraints. Can be described using either high-level or implementational data model.

• **external/view level**: includes a number of external schemas (or user views), each of which describes part of the database that a particular category of users is interested in, hiding rest of database. Can be described using either high-level or implementational data model. (In practice, usually described using same model as is the conceptual schema.)



 Page 23

DATABASE MANAGEMENT SYSTEM (21CS53)



Users (including application programs) submit queries that are expressed with respect to the external level. It is the responsibility of the DBMS to **transform** such a query into one that is expressed with respect to the internal level (and to transform the result, which is at the internal level, into its equivalent at the external level).

Example: Select students with GPA > 3.5.

Q:How is this accomplished? A: By virtue of **mappings** between the levels:

• **external/conceptual** mapping (providing **logical** data independence)

• **conceptual/internal** mapping (providing **physical** data independence)

**Data independence** is the capacity to change the schema at one level of the architecture without having to change the schema at the next higher level. We distinguish between **logical** and **physical** data independence according to which two adjacent levels are involved. The former refers to the ability to change the conceptual schema without changing the external schema. The latter refers to the ability to change the internal schema without having to change the conceptual.

For an **example of physical data independence**, suppose that the internal schema is modified (because we decide to add a new index, or change the encoding scheme used in representing some field's value, or stipulate that some previously unordered file must be ordered by a particular field ). Then we can change the mapping between the conceptual and internal schemas in order to avoid changing the conceptual schema itself.

Not surprisingly, the process of transforming data via mappings can be costly (performance wise), which is probably one reason that real-life DBMS's don't fully implement this 3-schema architecture.

**2.3 Database Languages and Interfaces**

A DBMS supports a variety of users and must provide appropriate languages and interfaces for each category of users.

**DBMS Languages**

• DDL (Data Definition Language): used (by the DBA and/or database designers) to specify the conceptual schema.

• SDL (Storage Definition Language): used for specifying the internal schema • VDL (View Definition Language): used for specifying the external schemas (i.e., user

Page 24

DATABASE MANAGEMENT SYSTEM (21CS53)



• DML (Data Manipulation Language): used for performing operations such as retrieval and update upon the populated database

The above description represents some kind of ideal. In real-life, at least so far, the de facto standard DBMS language is SQL (Standard Query Language), which has constructs to support the functions needed by DDL, VDL, and DML languages. (Early versions of SQL had features in support of SDL functions, but no more.)

**2.3.1 DBMS Languages**

menu-based, forms-based, gui-based, natural language, special purpose for parametric users, for DBA.

**2.3.2 DBMS Interfaces**

• Menu-based interfaces for web clients or browsing

• Forms-based interfaces

• GUI's

• Natural Language Interfaces

• Speech Input and Output

• Interfaces for parametric users

• Interfaces for the DBA

**2.4 Database System Environment**

See Figure 2.3, page 41.

**2.5 Centralized and Client/Server Architectures for DBMS's**

**2.6 Classification of DBMS's**

Based upon

• underlying data model (e.g., relational, object, object-relational, network) • multi-user vs. single-user

• centralized vs. distributed

• cost

• general-purpose vs. special-purpose

• types of **access path** options



Page 25

DATABASE MANAGEMENT SYSTEM (21CS53)



**2.7 Data Modeling Using the Entity-Relationship Model**

**Outline of Database Design**

The main phases of database design are depicted in Figure 3.1, page 59:

• **Requirements Collection and Analysis**: purpose is to produce a description of the users' requirements.

• **Conceptual Design**: purpose is to produce a *conceptual schema* for the database, including detailed descriptions of *entity types*, *relationship types*, and *constraints*. All these are expressed in terms provided by the data model being used. (*Remark*: As the ER model is focused on precisely these three concepts, it would seem that the authors are predisposed to using that data model!)

• **Implementation**: purpose is to transform the conceptual schema (which is at a high/abstract level) into a (lower-level) *representational/implementational* model supported by whatever DBMS is to be used.

• **Physical Design**: purpose is to decide upon the internal storage structures, access paths (indexes), etc., that will be used in realizing the representational model produced in previous phase.

**2.8: Entity-Relationship (ER) Model**

Our focus now is on the second phase, **conceptual design**, for which The **Entity-Relationship (ER) Model** is a popular high-level conceptual data model.

In the ER model, the main concepts are **entity**, **attribute**, and **relationship**. **2.8.1 Entities and Attributes**

**Entity**: An entity represents some "thing" (in the miniworld) that is of interest to us, i.e., about which we want to maintain some data. An entity could represent a physical object (e.g., house, person, automobile, widget) or a less tangible concept (e.g., company, job, academic course).

**Attribute**: An entity is described by its attributes, which are properties characterizing it. Each attribute has a **value** drawn from some **domain** (set of meaningful values).

Example: A *PERSON* entity might be described by *Name*, *BirthDate*, *Sex*, etc., attributes, each having a particular value.

What distinguishes an entity from an attribute is that the latter is strictly for the purpose of describing the former and is not, in and of itself, of interest to us. It is sometimes said that an



 Page 26

DATABASE MANAGEMENT SYSTEM (21CS53)



entity has an independent existence, whereas an attribute does not. In performing data modeling, however, it is not always clear whether a particular concept deserves to be classified as an entity or "only" as an attribute.

We can classify attributes along these dimensions:

• simple/atomic vs. composite

• single-valued vs. multi-valued (or set-valued)

• stored vs. derived (*Note from instructor:* this seems like an implementational detail that ought not be considered at this (high) level of abstraction.)

A **composite** attribute is one that is *composed* of smaller parts. An **atomic** attribute is indivisible or indecomposable.

• **Example 1**: A *BirthDate* attribute can be viewed as being composed of (sub-)attributes for month, day, and year.

• **Example 2**: An *Address* attribute (Figure 3.4, page 64) can be viewed as being composed of (sub-)attributes for street address, city, state, and zip code. A street address can itself be viewed as being composed of a number, street name, and apartment number. As this suggests, composition can extend to a depth of two (as here) or more.

To describe the structure of a composite attribute, one can draw a tree (as in the aforementioned Figure 3.4). In case we are limited to using text, it is customary to write its name followed by a parenthesized list of its sub-attributes. For the examples mentioned above, we would write

*BirthDate(Month, Day, Address(StreetAddr(StrNum, StrName, AptNum), City, State, Zip)*

*Year)*

**Single- vs. multi-valued** attribute: Consider a *PERSON* entity. The person it represents has (one) *SSN*, (one) *date of birth*, (one, although composite) *name*, etc. But that person may have zero or more academic degrees, dependents, or (if the person is a male living in Utah) spouses! How can we model this via attributes *AcademicDegrees*, *Dependents*, and *Spouses*? One way is to allow such attributes to be *multi-valued* (perhaps *set-valued* is a better term), which is to say that we assign to them a (possibly empty) *set* of values rather than a single value.

To distinguish a multi-valued attribute from a single-valued one, it is customary to enclose the former within curly braces (which makes sense, as such an attribute has a value that is a set, and curly braces are traditionally used to denote sets). Using the *PERSON* example from above, we would depict its structure in text as

*PERSON(SSN, Name, BirthDate(Month, Day, Year), { AcademicDegrees(School, Level, Year) }, { Dependents }, ...)*

**

** Page 27

DATABASE MANAGEMENT SYSTEM (21CS53)



Here we have taken the liberty to assume that each academic degree is described by a school, level (e.g., B.S., Ph.D.), and year. Thus, *AcademicDegrees* is not only multi-valued but also composite. We refer to an attribute that involves some combination of multi-valuedness *and* compositeness as a **complex** attribute.

A more complicated example of a complex attribute is *AddressPhone* in Figure 3.5 (page 65). This attribute is for recording data regarding addresses and phone numbers of a business. The structure of this attribute allows for the business to have several offices, each described by an address and a set of phone numbers that ring into that office. Its structure is given by

*{ AddressPhone( { Phone(AreaCode, Number) }, Address(StrAddr(StrNum, StrName, AptNum), City, State, Zip)) }*

**Stored vs. derived** attribute: Perhaps *independent* and *derivable* would be better terms for these (or *non-redundant* and *redundant*). In any case, a *derived* attribute is one whose value can be calculated from the values of other attributes, and hence need not be stored. **Example:** *Age* can be calculated from *BirthDate*, assuming that the current date is accessible.

**The Null value**: In some cases a particular entity might not have an applicable value for a particular attribute. Or that value may be unknown. Or, in the case of a multi-valued attribute, the appropriate value might be the empty set.

*Example*: The attribute *DateOfDeath* is not applicable to a living person and its correct value may be unknown for some persons who have died.

In such cases, we use a special attribute value (non-value?), called **null**. There has been some argument in the database literature about whether a different approach (such as having distinct values for *not applicable* and *unknown*) would be superior.

**2.8.2: Entity Types, Entity Sets, Keys, and Domains**

Above we mentioned the concept of a *PERSON* entity, i.e., a representation of a particular person via the use of attributes such as *Name*, *Sex*, etc. Chances are good that, in a database in which one such entity exists, we will want many others of the same kind to exist also, each of them described by the same collection of attributes. Of course, the *values* of those attributes will differ from one entity to another (e.g., one person will have the name "Mary" and another will have the name "Rumpelstiltskin"). Just as likely is that we will want our database to store information about other kinds of entities, such as business transactions or academic courses, which will be described by entirely different collections of attributes.

This illustrates the distinction between entity types and entity instances. An **entity type** serves as a template for a collection of **entity instances**, all of which are described by the same collection



Page 28

DATABASE MANAGEMENT SYSTEM (21CS53)



of attributes. That is, an entity type is analogous to a **class** in object-oriented programming and an entity instance is analogous to a particular object (i.e., instance of a class).

In ER modeling, we deal only with entity types, not with instances. In an ER diagram, each entity type is denoted by a rectangular box.

An **entity set** is the collection of all entities of a particular type that exist, in a database, at some moment in time.

**Key Attributes of an Entity Type**: A minimal collection of attributes (often only one) that, by design, distinguishes any two (simultaneously-existing) entities of that type. In other words, if attributes *A1* through *Am* together form a key of entity type *E*, and *e* and *f* are two entities of type *E* existing at the same time, then, in at least one of the attributes *Ai* (*0 < i <= m*), *e* and *f* must have distinct values.

An entity type could have more than one key. (An example of this appears in Figure 3.7, page 67, in which the CAR entity type is postulated to have both { *Registration(RegistrationNum, State)* } and { *VehicleID* } as keys.)

**Domains (Value Sets)** of Attributes: The domain of an attribute is the "universe of values" from which its value can be drawn. In other words, an attribute's domain specifies its set of allowable values. The concept is similar to **data type**.

**Example Database Application: COMPANY**

Suppose that Requirements Collection and Analysis results in the following (informal) description of the COMPANY miniworld:

The company is organized as a collection of **departments**.

• Each department

o has a unique name

o has a unique number

o is associated with a set of locations

o has a particular employee who acts as its manager (and who assumed that position on some date)

o has a set of employees assigned to it

o controls a set of projects

• Each project

o has a unique name

o has a unique number

o has a single location



Page 29

DATABASE MANAGEMENT SYSTEM (21CS53)



o has a set of employees who work on it

o is controlled by a single department

• Each employee

o has a name

o has a SSN that uniquely identifies her/him

o has an address

o has a salary

o has a sex

o has a birthdate

o has a direct supervisor

o has a set of dependents

o is assigned to one department

o works some number of hours per week on each of a set of projects (which need not all be controlled by the same department)

• Each dependent

o has first name

o has a sex

o has a birthdate

o is related to a particular employee in a particular way (e.g., child, spouse, pet) o is uniquely identified by the combination of her/his first name and the employee of which (s)he is a dependent

**2.8.3 Initial Conceptual Design of COMPANY database**

Using the above structured description as a guide, we get the following preliminary design for entity types and their attributes in the COMPANY database:

• DEPARTMENT(Name, Number, { Locations }, Manager, ManagerStartDate, { Employees }, { Projects })

• PROJECT(Name, Number, Location, { Workers }, ControllingDept)

• EMPLOYEE(Name(FName, MInit, LName), SSN, Sex, Address, Salary, BirthDate, Dept, Supervisor, { Dependents }, { WorksOn(Project, Hours) })

• DEPENDENT(Employee, FirstName, Sex, BirthDate, Relationship)

*Remarks*: Note that the attribute *WorksOn* of EMPLOYEE (which records on which projects the employee works) is not only multi-valued (because there may be several such projects) but also composite, because we want to record, for each such project, the number of hours per week that the employee works on it. Also, each *candidate key* has been indicated by underlining.

For similar reasons, the attributes *Manager* and *ManagerStartDate* of DEPARTMENT really ought to be combined into a single composite attribute. Not doing so causes little or no harm, however, because these are single-valued attributes. Multi-valued attributes would pose some



Page 30

DATABASE MANAGEMENT SYSTEM (21CS53)



difficulties, on the other hand. Suppose, for example, that a department could have two or more managers, and that some department had managers Mary and Harry, whose start dates were 10- 4-1999 and 1-13-2001, respectively. Then the values of the *Manager* and *ManagerStartDate* attributes should be *{ Mary, Harry }* and *{ 10-4-1999, 1-13-2001 }*. But from these two attribute values, there is no way to determine which manager started on which date. On the other hand, by recording this data as a set of ordered pairs, in which each pair identifies a manager and her/his starting date, this deficiency is eliminated. *End of Remarks*

**2.9 Relationship Types, Sets, Roles, and Structural Constraints**

Having presented a preliminary database schema for COMPANY, it is now convenient to clarify the concept of a **relationship** (which is the last of the three main concepts involved in the ER model).

**Relationship**: This is an association between two entities. As an example, one can imagine a STUDENT entity being associated to an ACADEMIC\_COURSE entity via, say, an ENROLLED\_IN relationship.

Whenever an attribute of one entity type refers to an entity (of the same or different entity type), we say that a relationship exists between the two entity types.

From our preliminary COMPANY schema, we identify the following **relationship types** (using descriptive names and ordering the participating entity types so that the resulting phrase will be in active voice rather than passive):

• EMPLOYEE MANAGES DEPARTMENT (arising from *Manager* attribute in DEPARTMENT)

• DEPARTMENT CONTROLS PROJECT (arising from *ControllingDept* attribute in PROJECT and the *Projects* attribute in DEPARTMENT)

• EMPLOYEE WORKS\_FOR DEPARTMENT (arising from *Dept* attribute in EMPLOYEE and the *Employees* attribute in DEPARTMENT)

• EMPLOYEE SUPERVISES EMPLOYEE (arising from *Supervisor* attribute in EMPLOYEE)

• EMPLOYEE WORKS\_ON PROJECT (arising from *WorksOn* attribute in EMPLOYEE and the *Workers* attribute in PROJECT)

• DEPENDENT DEPENDS\_ON EMPLOYEE (arising from *Employee* attribute in DEPENDENT and the *Dependents* attribute in EMPLOYEE)

In ER diagrams, relationship types are drawn as diamond-shaped boxes connected by lines to the entity types involved. See Figure 3.2, page 62. Note that attributes are depicted by ovals connected by lines to the entity types they describe (with multi-valued attributes in double ovals and composite attributes depicted by trees). The original attributes that gave rise to the relationship types are absent, having been replaced by the relationship types.

Page 31

DATABASE MANAGEMENT SYSTEM (21CS53)



A **relationship set** is a set of instances of a relationship type. If, say, R is a relationship type that relates entity types A and B, then, at any moment in time, the relationship set of R will be a set of ordered pairs *(x,y)*, where *x* is an instance of A and *y* is an instance of B. What this means is that, for example, if our COMPANY miniworld is, at some moment, such that employees *e1*, *e3*, and *e6* work for department *d1*, employees *e2* and *e4* work for department *d2*, and employees *e5* and *e7*

work for department *d3*, then the **relationship set** will include as **instances** the WORKS\_FOR

ordered pairs (*e1*, *d1*), (*e2*, *d2)*, (*e3*, *d1)*, (*e4*, *d2)*, (*e5*, *d3)*, (*e6*, *d1)*, and (*e7*, *d3)*. See Figure 3.9 on page 71 for a graphical depiction of this.

**2.9.1 Ordering of entity types in relationship types**: Note that the order in which we list the entity types in describing a relationship is of little consequence, except that the relationship name (for purposes of clarity) ought to be consistent with it. For example, if we swap the two entity

types in each of the first two relationships listed above, we should rename them IS\_MANAGED\_BY and IS\_CONTROLLED\_BY, respectively.

**2.9.2 Degree of a relationship type**: Also note that, in our COMPANY example, all relationship instances will be ordered pairs, as each relationship associates an instance from one entity type with an instance of another (or the same, in the case of SUPERVISES) relationship type. Such relationships are said to be *binary*, or to have *degree* two. Relationships with degree three (called *ternary*) or more are also possible, although not as common. This is illustrated in Figure 3.10 (page 72), where a relationship SUPPLY (perhaps not the best choice for a name) has as instances ordered triples of suppliers, parts, and projects, with the intent being that inclusion of the ordered triple *(s2, p4, j1)*, for example, indicates that supplier *s2* supplied part *p4* to project *j1)*.

**Roles in relationships**: Each entity that participates in a relationship plays a particular *role* in that relationship, and it is often convenient to refer to that role using an appropriate name. For example, in each instance of a WORKS\_FOR relationship set, the employee entity plays the role of *worker* or (surprise!) *employee* and each department plays the role of *employer* or (surprise!) *department*. Indeed, as this example suggests, often it is best to use the same name for the role as for the corresponding entity type.

An exception to this rule occurs when the same entity type plays two (or more) roles in the same relationship. (Such relationships are said to be *reCURsive*, which I find to be a misleading use of that term. A better term might be *self-referential*.) For example, in each instance of a SUPERVISES relationship set, one employee plays the role of *supervisor* and the other plays the role of *supervisee*.

**2.9.3 Constraints on Relationship Types**

Often, in order to make a relationship type be an accurate model of the miniworld concepts that it is intended to represent, we impose certain constraints that limit the possible corresponding relationship sets. (That is, a constraint may make "invalid" a particular set of instances for a relationship type.)



 Page 32

DATABASE MANAGEMENT SYSTEM (21CS53)

There are two main kinds of relationship constraints (on binary relationships). For illustration, let *R* be a relationship set consisting of ordered pairs of instances of entity types *A* and *B*, respectively.

• **cardinality ratio**:

o **1:1 (one-to-one)**: Under this constraint, no instance of *A* may particpate in more than one instance of *R*; similarly for instances of *B*. In other words, if *(a1, b1)* and *(a2, b2)* are (distinct) instances of *R*, then neither *a1 = a2* nor *b1 = b2*. **Example**: Our informal description of COMPANY says that every department has one employee who manages it. If we also stipulate that an employee may not

(simultaneously) play the role of manager for more than one department, it follows that MANAGES is 1:1.

o **1:N (one-to-many)**: Under this constraint, no instance of *B* may participate in more than one instance of *R*, but instances of *A* are under no such restriction. In other words, if *(a1, b1)* and *(a2, b 2)* are (distinct) instances of *R*, then it cannot be the case that *b1 = b2*. **Example**: CONTROLS is 1:N because no project may be controlled by more than one department. On the other hand, a department may control any number of projects, so there is no restriction on the number of relationship instances in which a particular department instance may participate. For similar reasons, SUPERVISES is also 1:N.

o **N:1 (many-to-one)**: This is just the same as 1:N but with roles of the two entity types reversed. **Example**: WORKS\_FOR and DEPENDS\_ON are N:1.

o **M:N (many-to-many)**: Under this constraint, there are no restrictions. (Hence, the term applies to the absence of a constraint!) **Example**: WORKS\_ON is M:N, because an employee may work on any number of projects and a project may have any number of employees who work on it.

Notice the notation in Figure 3.2 for indicating each relationship type's cardinality ratio.

Suppose that, in designing a database, we decide to include a binary relationship *R* as described above (which relates entity types *A* and *B*, respectively). To determine how *R* should be constrained, with respect to cardinality ratio, the questions you should ask are these:

May a given entity of type B be related to multiple entities of type A? May a given entity of type A be related to multiple entities of type B?

Page 34

DATABASE MANAGEMENT SYSTEM (21CS53)



• **participation**: specifies whether or not the existence of an entity depends upon its being related to another entity via the relationship.

o **total participation (or existence dependency)**: To say that entity type *A* is constrained to **participate totally** in relationship *R* is to say that if (at some moment in time) *R*'s instance set is

*{ (a1, b1), (a2, b2), ... (am, bm) }*,

then (at that same moment) *A*'s instance set must be *{ a1, a2, ..., am }*. In other words, there can be no member of *A*'s instance set that does not participate in at least one instance of *R*.

According to our informal description of COMPANY, every employee must be assigned to some department. That is, every employee instance must participate in at least one instance of WORKS\_FOR, which is to say that *EMPLOYEE* satisfies the total participation constraint with respect to the WORKS\_FOR relationship.

In an ER diagram, if entity type *A* must participate totally in relationship type R, the two are connected by a double line. See Figure 3.2.

o **partial participation**: the absence of the total participation constraint! (E.g., not every employee has to participate in MANAGES; hence we say that, with respect to MANAGES, *EMPLOYEE* participates partially. This is not to say that for all employees to be managers is not allowed; it only says that it need not be the case that all employees are managers.

**2.9.4 Attributes of Relationship Types** (page 76)

Relationship types, like entity types, can have attributes. A good example is WORKS\_ON, each instance of which identifies an employee and a project on which (s)he works. In order to record (as the specifications indicate) how many hours are worked by each employee on each project, we include *Hours* as an attribute of WORKS\_ON. (See Figure 3.2 again.) In the case of an M:N relationship type (such as WORKS\_ON), allowing attributes is vital. In the case of an N:1, 1:N, or 1:1 relationship type, any attributes can be assigned to the entity type opposite from the 1 side. For example, the *StartDate* attribute of the MANAGES relationship type can be given to either the *EMPLOYEE* or the *DEPARTMENT* entity type.



 Page 34

DATABASE MANAGEMENT SYSTEM (21CS53)



**2.10 Weak Entity Types**: An entity type that has no set of attributes that qualify as a key is called **weak**. (Ones that do are **strong**.)

An entity of a weak identity type is uniquely identified by the specific entity to which it is related (by a so-called **identifying relationship** that relates the weak entity type with its so-called **identifying** or **owner entity type**) in combination with some set of its own attributes (called a *partial key*).

**Example**: A *DEPENDENT* entity is identified by its first name together with the *EMPLOYEE* entity to which it is related via DEPENDS\_ON. (Note that this wouldn't work for former heavyweight boxing champion George Foreman's sons, as they all have the name "George"!)

Because an entity of a weak entity type cannot be identified otherwise, that type has a **total participation constraint** (i.e., **existence dependency**) with respect to the identifying relationship.

This should not be taken to mean that any entity type on which a total participation constraint exists is weak. For example, DEPARTMENT has a total participation constraint with respect to MANAGES, but it is not weak.

In an ER diagram, a weak entity type is depicted with a double rectangle and an identifying relationship type is depicted with a double diamond.

**Design Choices for ER Conceptual Design**: Sometimes it is not clear whether a particular miniworld concept ought to be modeled as an entity type, an attribute, or a relationship type. Here are some guidelines (given with the understanding that schema design is an iterative process in which an initial design is refined repeatedly until a satisfactory result is achieved):

• As happened in our development of the ER model for COMPANY, if an attribute of entity type *A* serves as a reference to an entity of type *B*, it may be wise to refine that attribute into a binary relationship involving entity types *A* and *B*. It may well be that *B* has a corresponding attribute referring back to *A*, in which case it, too, is refined into the aforementioned relationship. In our COMPANY example, this was exemplified by the *Projects* and *ControllingDept* attributes of *DEPARTMENT* and *PROJECT*, respectively.

• An attribute that exists in several entity types may be refined into its own entity type. For example, suppose that in a UNIVERSITY database we have entity types *STUDENT*, *INSTRUCTOR*, and *COURSE*, all of which have a *Department* attribute. Then it may be wise to introduce a new entity type, *DEPARTMENT*, and then to follow the preceding guideline by introducing a binary relationship between *DEPARTMENT* and each of the three aforementioned entity types.

• An entity type that is involved in very few relationships (say, zero, one, or possibly two) could be refined into an attribute (of each entity type to which it is related).



 Page 35

DATABASE MANAGEMENT SYSTEM (21CS53)



**Questions**

1. Design an ER Diagram for keeping track of Information about Bank Database,Taking into account 4 entities?

2. Describe how to map the following Scenario‘s in ER Model to schema,with suitable exam ple:

3. List the summary of the notations for ER diagrams. Include symbols used in ER diagram and their meaning.

4. With respect to ER model explain with example.

5. What is meant by partial key? Explain.

6. Define an entity and an attribute,explain the different types of attributes that occur in an ER diagram model,with an example

7. Define the following with an example

i. Weak entity types

ii. Cardinality ratio

iii. Ternary relationship

iv. Participation constraints



 Page 36

DATABASE MANAGEMENT SYSTEM (21CS53)



**The Relational Data Model and Relational Database Constraints**

Subject Code : **21CS53** Exam Hours : 03

IA Marks : 25

Total No. of Lecture Hours : 50

No. of Lecture Hours/Week : 04 Exam Marks : 80

**The Relational Data Model and Relational Database Constraints and Relational Algebra** 3.1 Relational Model Concepts

3.1.2 Characteristics of Relations

3.1.3 Relational Model Notation

3.2 Relational Model Constraints and Relational Database Schemas

3.2.1 Domain Constraints

3.2.2 Key Constraints

3.2.3 Relational Databases and Relational Database Schemas

3.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

3.3 Update Operations and Dealing with Constraint Violations

3.3.1 Insert

3.3.2 Delete

3.3.3 Update:

3.3.4 Transactions and dealing with constraints

3.4 Relational Operation

3.5 Relational algebra operation Set theory Operations

3.6 JOIN Operations

3.7 Additional Relational Operations

3.8 Examples of Queries in Relational Algebra

3.9 Relational Database Design Using ER-to-Relational Mapping



 Page 37

DATABASE MANAGEMENT SYSTEM (21CS53)



**The Relational Data Model and Relational Database Constraints and Relational Algebra**

**Origins**

**3.1 Relational Model Concepts**

• **Domain**: A (usually named) set/universe of *atomic* values, where by "atomic" we mean simply that, from the point of view of the database, each value in the domain is indivisible (i.e., cannot be broken down into component parts).

Examples of domains (some taken from page 147):

o USA\_phone\_number: string of digits of length ten

o SSN: string of digits of length nine

o Name: string of characters beginning with an upper case letter

o GPA: a real number between 0.0 and 4.0

o Sex: a member of the set { female, male }

o Dept\_Code: a member of the set { CMPS, MATH, ENGL, PHYS, PSYC, ... }

These are all *logical* descriptions of domains. For implementation purposes, it is necessary to provide descriptions of domains in terms of concrete **data types** (or **formats**) that are provided by the DBMS (such as String, int, boolean), in a manner analogous to how programming languages have intrinsic data types.

• **Attribute**: the *name* of the role played by some value (coming from some domain) in the context of a **relational schema**. The domain of attribute A is denoted dom(A). • **Tuple**: A tuple is a mapping from attributes to values drawn from the respective domains of those attributes. A tuple is intended to describe some entity (or relationship between entities) in the miniworld.

As an example, a tuple for a PERSON entity might be

{ Name --> "Rumpelstiltskin", Sex --> Male, IQ --> 143 }

• **Relation**: A (named) set of tuples all of the same form (i.e., having the same set of attributes). The term **table** is a loose synonym. (Some database purists would argue that a table is "only" a physical manifestation of a relation.)

• **Relational Schema**: used for describing (the structure of) a relation. E.g., R(A1, A2, ..., An) says that R is a relation with *attributes* A1, ... An. The **degree** of a relation is the number of attributes it has, here *n*.

Example: STUDENT(Name, SSN, Address)

Page 38

DATABASE MANAGEMENT SYSTEM (21CS53)



(See Figure 5.1, page 149, for an example of a STUDENT relation/table having several tuples/rows.)

One would think that a "complete" relational schema would also specify the domain of each attribute.

• **Relational Database**: A collection of **relations**, each one consistent with its specified relational schema.

**3.1.2 Characteristics of Relations**

**Ordering of Tuples**: A relation is a *set* of tuples; hence, there is no order associated with them. That is, it makes no sense to refer to, for example, the 5th tuple in a relation. When a relation is depicted as a table, the tuples are necessarily listed in *some* order, of course, but you should attach no significance to that order. Similarly, when tuples are represented on a storage device, they must be organized in *some* fashion, and it may be advantageous, from a performance standpoint, to organize them in a way that depends upon their content.

**Ordering of Attributes**: A tuple is best viewed as a mapping from its attributes (i.e., the names we give to the roles played by the values comprising the tuple) to the corresponding values. Hence, the order in which the attributes are listed in a table is irrelevant. (Note that, unfortunately, the set theoretic operations in relational algebra (at least how E&N define them) make implicit use of the order of the attributes. Hence, E&N view attributes as being arranged as a sequence rather than a set.)

**Values of Attributes**: For a relation to be in *First Normal Form*, each of its attribute domains must consist of atomic (neither composite nor multi-valued) values. Much of the theory underlying the relational model was based upon this assumption. Chapter 10 addresses the issue of including non-atomic values in domains. (Note that in the latest edition of C.J. Date's book, he explicitly argues against this idea, admitting that he has been mistaken in the past.)

The **Null** value: used for *don't know*, *not applicable*.

**Interpretation of a Relation**: Each relation can be viewed as a **predicate** and each tuple in that relation can be viewed as an assertion for which that predicate is satisfied (i.e., has value **true**) for the combination of values in it. In other words, each tuple represents a fact. Example (see Figure 5.1): The first tuple listed means: There exists a student having name Benjamin Bayer, having SSN 305-61-2435, having age 19, etc.

Keep in mind that some relations represent facts about entities (e.g., students) whereas others represent facts about relationships (between entities). (e.g., students and course sections).

The **closed world assumption** states that the only true facts about the miniworld are those represented by whatever tuples currently populate the database.



 Page 39

DATABASE MANAGEMENT SYSTEM (21CS53)



**3.1.3 Relational Model Notation**: page 152

• *R(A1, A2, ..., An)* is a relational schema of degree *n* denoting that there is a relation *R* having as its attributes *A1, A2, ..., An*.

• By convention, *Q*, *R*, and *S* denote relation names.

• By convention, *q*, *r*, and *s* denote relation states. For example, *r(R)* denotes one possible state of relation *R*. If *R* is understood from context, this could be written, more simply, as *r*.

• By convention, *t*, *u*, and *v* denote tuples.

• The "dot notation" *R.A* (e.g., STUDENT.Name) is used to qualify an attribute name, usually for the purpose of distinguishing it from a same-named attribute in a different relation (e.g., DEPARTMENT.Name).

•

**3.2 Relational Model Constraints and Relational Database Schemas** Constraints on databases can be categorized as follows:

• **inherent model-based:** Example: no two tuples in a relation can be duplicates (because a relation is a set of tuples)

• **schema-based:** can be expressed using DDL; this kind is the focus of this section. • **application-based:** are specific to the "business rules" of the miniworld and typically difficult or impossible to express and enforce within the data model. Hence, it is left to application programs to enforce.

Elaborating upon **schema-based constraints**:

**3.2.1 Domain Constraints**: Each attribute value must be either **null** (which is really a *non-value*) or drawn from the domain of that attribute. Note that some DBMS's allow you to impose the **not null** constraint upon an attribute, which is to say that that attribute may not have the (non-)value **null**.

**3.2.2 Key Constraints**: A relation is a *set* of tuples, and each tuple's "identity" is given by the values of its attributes. Hence, it makes no sense for two tuples in a relation to be identical (because then the two tuples are actually one and the same tuple). That is, no two tuples may have the same combination of values in their attributes.

Usually the miniworld dictates that there be (proper) subsets of attributes for which no two tuples may have the same combination of values. Such a set of attributes is called a **superkey** of its relation. From the fact that no two tuples can be identical, it follows that the set of all attributes of a relation constitutes a superkey of that relation.

A **key** is a *minimal superkey*, i.e., a superkey such that, if we were to remove any of its attributes, the resulting set of attributes fails to be a superkey.



Mr. MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 40

DATABASE MANAGEMENT SYSTEM (21CS53)



**Example**: Suppose that we stipulate that a faculty member is uniquely identified by *Name* and *Address* and also by *Name* and *Department*, but by no single one of the three attributes mentioned. Then *{ Name, Address, Department }* is a (non-minimal) superkey and each of *{ Name, Address }* and *{ Name, Department }* is a key (i.e., minimal superkey).

**Candidate key**: any key! (Hence, it is not clear what distinguishes a key from a candidate key.)

**Primary key**: a key chosen to act as the means by which to identify tuples in a relation. Typically, one prefers a primary key to be one having as few attributes as possible.

**3.2.3 Relational Databases and Relational Database Schemas**

A **relational database schema** is a set of schemas for its relations (see Figure 5.5, page 157) together with a set of **integrity constraints**.

A **relational database state/instance/snapshot** is a set of states of its relations such that no integrity constraint is violated. (See Figure 5.6, page 159, for a snapshot of COMPANY.)

**3.2.4 Entity Integrity, Referential Integrity, and Foreign Keys**

**Entity Integrity Constraint**: In a tuple, none of the values of the attributes forming the relation's primary key may have the (non-)value **null**. Or is it that at least one such attribute must have a non-null value? In my opinion, E&N do not make it clear!

**Referential Integrity Constraint**: (See Figure 5.7) A **foreign key** of relation *R* is a set of its attributes intended to be used (by each tuple in *R*) for identifying/referring to a tuple in some relation *S*. (*R* is called the *referencing* relation and *S* the *referenced* relation.) For this to make sense, the set of attributes of *R* forming the foreign key should "correspond to" some superkey of *S*. Indeed, by definition we require this superkey to be the primary key of *S*.

This constraint says that, for every tuple in *R*, the tuple in *S* to which it refers must actually be in *S*. Note that a foreign key may refer to a tuple in the same relation and that a foreign key may be part of a primary key (indeed, for weak entity types, this will always occur). A foreign key may have value **null** (necessarily in all its attributes??), in which case it does not refer to any tuple in the referenced relation.

**Semantic Integrity Constraints**: application-specific restrictions that are unlikely to be expressible in DDL. Examples:

• salary of a supervisee cannot be greater than that of her/his supervisor

• salary of an employee cannot be lowered

**3.3 Update Operations and Dealing with Constraint Violations**

****

**** Page 41

DATABASE MANAGEMENT SYSTEM (21CS53)



For each of the *update* operations (Insert, Delete, and Update), we consider what kinds of constraint violations may result from applying it and how we might choose to react.

**3.3.1 Insert**:

• domain constraint violation: some attribute value is not of correct domain • entity integrity violation: key of new tuple is **null**

• key constraint violation: key of new tuple is same as existing one

• referential integrity violation: foreign key of new tuple refers to non-existent tuple

Ways of dealing with it: reject the attempt to insert! Or give user opportunity to try again with different attribute values.

**3.3.2 Delete**:

• referential integrity violation: a tuple referring to the deleted one exists. Three options for dealing with it:

• Reject the deletion

• Attempt to **cascade** (or **propagate**) by deleting any referencing tuples (plus those that reference them, etc., etc.)

• modify the foreign key attribute values in referencing tuples to **null** or to some valid value referencing a different tuple

**3.3.3 Update**:

• Key constraint violation: primary key is changed so as to become same as another tuple's • referential integrity violation:

o foreign key is changed and new one refers to nonexistent tuple

o primary key is changed and now other tuples that had referred to this one violate the constraint

**3.3.4 Transactions**: This concept is relevant in the context where multiple users and/or application programs are accessing and updating the database concurrently. A transaction is a logical unit of work that may involve several accesses and/or updates to the database (such as what might be required to reserve several seats on an airplane flight). The point is that, even though several transactions might be processed concurrently, the end result must be as though the transactions were carried out sequentially. (Example of simultaneous withdrawals from same checking account.)



 Page 42

DATABASE MANAGEMENT SYSTEM (21CS53)



**The Relational Algebra**

• Operations to manipulate relations.

• Used to specify retrieval requests (queries).

• Query result is in the form of a relation

**3.4 Relational Operations: SELECT ( ) and Project π operations**

Set operations: These include UNION U, INTERSECTION | |, DIFFERENCE -, CARTESIAN PRODUCT X.

JOIN operations .

Other relational operations: DIVISION, OUTER JOIN, AGGREGATE FUNCTIONS. **3.4.1 SELECT and PROJECT **

**SELECT** operation (denoted by ):

• Selects the tuples (rows) from a relation R that satisfy a certain *selection condition* c

• Form of the operation: c

• The condition c is an arbitrary Boolean expression on the attributes of R

• Resulting relation has the *same attributes* as R

• Resulting relation includes each tuple in r(R) whose attribute values satisfy the condition c

Examples:

DNO=4(EMPLOYEE)

(EMPLOYEE)

SALARY>30000

(EMPLOYEE)

(DNO=4 AND SALARY>25000) OR DNO=5

 Page 43

DATABASE MANAGEMENT SYSTEM (21CS53)



**PROJECT** operation (denoted by ):

Keeps only certain

attributes (columns) from a relation R specified in an *attribute list* L

• Form of

operation: L(R)

• Resulting relation has only those attributes of R specified in L

• The PROJECT operation eliminates duplicate tuples in the resulting relation so that it remains a mathematical set (no duplicate elements).

Example: SEX,SALARY(EMPLOYEE)

If several male employees have salary 30000, only a single tuple <M, 30000> is kept in the resulting relation.

**Duplicate tuples are eliminated by the operation**.

**Sequences of operations**:



 Page 44

DATABASE MANAGEMENT SYSTEM (21CS53)



Several operations can be combined to form a *relational algebra expression* (query) Example: Retrieve the names and salaries of employees who work in department 4:

FNAME,LNAME,SALARY ( DNO=4(EMPLOYEE) )

*Alternatively*, we specify explicit intermediate relations for each

step:

DEPT4\_EMPS DNO=4(EMPLOYEE)

π

FNAME,LNAME,SALARY

(DEPT4\_EMPS)

Attributes can optionally be *renamed* in the resulting left-hand-side relation (this may be required for some operations that will be presented later):

DEPT4\_EMPS DNO=4(EMPLOYEE)

(FIRSTNAME,LASTNAME,SALARY) FNAME,LNAME,SALARY(DEPT4\_EMPS) **3.5 Relational algebra operation Set theory Operations**

****

**** Page 45

DATABASE MANAGEMENT SYSTEM (21CS53)



Binary operations from mathematical set theory:

**UNION**: R1 R2,

**INTERSECTION**: R1 R2,

**SET DIFFERENCE**: R1 - R2,

**CARTESIAN PRODUCT**: R1 X R2.

For , , -, the operand relations R1(A1, A2, ..., An) and R2(B1, B2, ..., Bn) must have the same number of attributes, and the domains of corresponding attributes must be compatible; that is, dom(Ai) = dom(Bi) for i=1, 2, ..., n. This condition is called union compatibility. The resulting relation for , , or - has the same attribute names as the first operand relation R1 (by convention).

**CARTESIAN PRODUCT**

R(A1, A2, ..., Am, B1, B2, ..., Bn) R1(A1, A2, ..., Am) X R2 (B1, B2, ..., Bn)



 Page 46

DATABASE MANAGEMENT SYSTEM (21CS53)



A tuple t exists in R for each combination of tuples t1 from R1 and

t2 from R2 such that:

t[A1, A2, ..., Am] = t1 and t[B1, B2, ..., Bn] = t2

If R1 has n1 tuples and R2 has n2 tuples, then R will have n1\*n2 tuples.

CARTESIAN PRODUCT is a *meaningless operation* on its own. It can *combine related tuples* from two relations *if followed by the appropriate SELECT operation*.

Example: Combine each DEPARTMENT tuple with the EMPLOYEE tuple of the manager. DEP\_EMP DEPARTMENT X EMPLOYEE

DEPT\_MANAGER MGRSSN=SSN(DEP\_EMP)

**3.6 JOIN Operations**

**THETA JOIN**: Similar to a CARTESIAN PRODUCT followed by a SELECT. The condition c is called a *join condition*.



 Page 47

DATABASE MANAGEMENT SYSTEM (21CS53)



R(A1, A2, ..., Am, B1, B2, ..., Bn) R1(A1, A2, ..., Am) c R2 (B1, B2, ..., Bn)

**EQUIJOIN**: The join condition c includes one or more *equality comparisons* involving attributes from R1 and R2. That is, c is of the form:

(Ai=Bj) AND ... AND (Ah=Bk); 1<i,h<m, 1<j,k<n

In the above EQUIJOIN operation:

Ai, ..., Ah are called the **join attributes** of R1

Bj, ..., Bk are called the **join attributes** of R2

Example of using EQUIJOIN:

Retrieve each DEPARTMENT's name and its manager's name:

T DEPARTMENT MGRSSN = SSN EMPLOYEE

RESULT  

DNAME,FNAME,LNAME

**NATURAL JOIN** (\*):

(T)

In an EQUIJOIN R R1 c R2, the join attribute of R2 appear *redundantly* in the result

relation R. In a NATURAL JOIN, the *redundant join attributes* of R2 are *eliminated* from R.  The equality condition is *implied* and need not be specified.

R R1 \*(join attributes of R1),(join attributes of R2)R2

Example: Retrieve each EMPLOYEE's name and the name of the DEPARTMENT he/she works for:

T EMPLOYEE \*(DNO),(DNUMBER) DEPARTMENT

RESULT 

FNAME,LNAME,DNAME

(T)

If the join attributes *have the same names* in both relations, they *need not be specified* and we can write R R1 \* R2.

Example: Retrieve each EMPLOYEE's name and the name of his/her SUPERVISOR:



 Page 48

DATABASE MANAGEMENT SYSTEM (21CS53)

SUPERVISOR(SUPERSSN,SFN,SLN) π (EMPLOYEE)

T EMPLOYEE \* SUPERVISOR

SSN,FNAME,LNAME

RESULT 

FNAME,LNAME,SFN,SLN

(T)

Note: In the *original definition* of NATURAL JOIN, the join attributes were *required* to have the same names in both relations.

There can be a *more than one set of join attributes* with a *different meaning* between the same two relations. For example:

JOIN ATTRIBUTES

RELATIONSHIP

EMPLOYEE.SSN= EMPLOYEE *manages*

DEPARTMENT.MGRSSN

the DEPARTMENT

EMPLOYEE.DNO= EMPLOYEE *works for* DEPARTMENT.DNUMBER the DEPARTMENT



 Page 49

DATABASE MANAGEMENT SYSTEM (21CS53)



Example: Retrieve each EMPLOYEE's name and the name of the DEPARTMENT he/she works for:

T EMPLOYEE DNO=DNUMBER DEPARTMENT

RESULT  

FNAME,LNAME,DNAME

(T)

A relation can have a *set of join attributes* to join it with *itself* :

JOIN ATTRIBUTES RELATIONSHIP

EMPLOYEE(1).SUPERSSN= EMPLOYEE(2) *supervises* EMPLOYEE(2).SSN EMPLOYEE(1)

One can *think of this* as joining *two distinct copies* of the relation, although only one relation actually exists In this case, *renaming* can be useful.

Example: Retrieve each EMPLOYEE's name and the name of his/her SUPERVISOR: SUPERVISOR(SSSN,SFN,SLN) SSN,FNAME,LNAME(EMPLOYEE) T EMPLOYEE SUPERSSN=SSSNSUPERVISOR

RESULT

FNAME,LNAME,SFN,SLN

(T)

**Complete Set of Relational Algebra Operations**:

All the operations discussed so far can be described as a sequence of *only* the operations SELECT, PROJECT, UNION, SET DIFFERENCE, and CARTESIAN PRODUCT. Hence, the set { , , , - , X } is called a *complete set* of relational algebra operations. Any query language *equivalent to* these operations is called **relationally complete**.

For database applications, additional operations are needed that were not part of the *original* relational algebra. These include:

1. Aggregate functions and grouping.

2. OUTER JOIN and OUTER UNION.

**3.7 Additional Relational Operations**

****

**** Page 50

DATABASE MANAGEMENT SYSTEM (21CS53)



**AGGREGATE FUNCTIONS ( )**

Functions such as SUM, COUNT, AVERAGE, MIN, MAX are often applied to sets of values or sets of tuples in database applications

<grouping attributes> <function list>(R)

The grouping attributes are optional

Example 1: Retrieve the average salary of all employees (no grouping needed): (AVGSAL) AVERAGE SALARY (EMPLOYEE)

Example 2: For each department, retrieve the department number, the number of employees, and the average salary (in the department):

(DNO,NUMEMPS,AVGSAL) DNO COUNT SSN, AVERAGE SALARY (EMPLOYEE) DNO is called the *grouping attribute* in the above example

**OUTER JOIN**

In a regular EQUIJOIN or NATURAL JOIN operation, tuples in R1 or R2 that do not have  matching tuples in the other relation *do not appear in the result*

Some queries require all tuples in R1 (or R2 or both) to appear in

the result

When no matching tuples are found, **null**s are placed for the

missing attributes

**LEFT OUTER JOIN**: R1 X R2 lets every tuple in R1 appear in the result

DATABASE MANAGEMENT SYSTEM (21CS53)



**RIGHT OUTER JOIN**: R1 X R2 lets every tuple in R2 appear in the result **FULL OUTER JOIN**: R1 X R2 lets every tuple in R1 or R2 appear in the result 

**3.8 Examples of Queries in Relational Algebra**

⚫**Q1: Retrieve the name and address of all employees who work for the ‘Research’ department.** RESEARCH\_DEPT ← σ DNAME=‘Research‘ (DEPARTMENT)

RESEARCH\_EMPS← (RESEARCH\_DEPT DNUMBER= DNOEMPLOYEEEMPLOYEE)

RESULT ← π FNAME, LNAME, ADDRESS (RESEARCH\_EMPS)

⚫**Q6: Retrieve the names of employees who have no dependents.**

ALL\_EMPS ← π SSN(EMPLOYEE)

EMPS\_WITH\_DEPS(SSN) ← π ESSN(DEPENDENT)

EMPS\_WITHOUT\_DEPS ← (ALL\_EMPS - EMPS\_WITH\_DEPS)

Page 52

DATABASE MANAGEMENT SYSTEM (21CS53)

RESULT ← π LNAME, FNAME (EMPS\_WITHOUT\_DEPS \* EMPLOYEE)

**3.9 Relational Database Design Using ER-to-Relational Mapping**

Step 1: For each **regular (strong) entity type** E in the ER schema, create a relation R that includes all the simple attributes of E.



EMPLOYEE

| SSNLnameFname |
| --- |

DEPARTMENT

| NUMBERNAME |
| --- |

Step 2: For each **weak entity type** W in the ER schema with owner entity type E, create a relation R, and include all simple attributes (or simple components of composite attributes) of W as attributes. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).

Page 53

DATABASE MANAGEMENT SYSTEM (21CS53)



DEPENDENT

| EMPL-SSNNAMERelationship |
| --- |

Step 3: For each **binary 1:1 relationship type** R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. Choose one of the relations, say S, and include the primary key of T as a foreign key in S. Include all the simple attributes of R as attributes of S.



DEPARTMENT

| MANAGER-SSN StartDate |
| --- |

Step 4: For each regular **binary 1:N relationship type** R identify the relation (N) relation S. Include the primary key of T as a foreign key of S. Simple attributes of R map to attributes of S.

Page 54

DATABASE MANAGEMENT SYSTEM (21CS53)



EMPLOYEE

| SupervisorSSN |
| --- |

Step 5: For each **binary M:N relationship type** R, create a relation S. Include the primary keys of participant relations as foreign keys in S. Their combination will be the primary key for S. Simple attributes of R become attributes of S.



WORKS-FOR

| EmployeeSSNDeptNumber |
| --- |

Step 6: For each **multi-valued attribute A**, create a new relation R. This relation will include an attribute corresponding to A, plus the primary key K of the parent relation (entity type or relationship type) as a foreign key in R. The primary key of R is the combination of A and K.



Page 55

DATABASE MANAGEMENT SYSTEM (21CS53)





DEP-LOCATION

| LocationDEP-NUMBER |
| --- |

Step 7: For each **n-ary relationship type R**, where n>2, create a new relation S to represent R. Include the primary keys of the relations participating in R as foreign keys in S. Simple attributes of R map to attributes of S. The primary key of S is a combination of all the foreign keys that reference the participants that have cardinality constraint > 1.

For a recursive relationship, we will need a new relation.



Page 56

DATABASE MANAGEMENT SYSTEM (21CS53)



**Questions**

1. Define the following terms with an example for each.

2. Explain:

3. i) Domain constraint ii) Semantic integrity constraint iii) Functional dependency constraint

4. List the characteristics of relation? Discuss any one?

5. Discuss various types of Inner Join Operations?

6. Discuss the characteristics of a relation, with an example

7. Briefly discuss the different types of update operations on relational database. show an example of

8. What is valid state and an invalid state,with respect to a database

9. Define referential integrity constraint. Explain the importance of referential integrity constraint. How is this constraint implemented in SQL

10. Define referential integrity in each of the update operation

**MODULE 2**

Page 57

DATABASE MANAGEMENT SYSTEM (21CS53)

**SQL The Relational Database Standard**

Subject Code : **21CS53** Exam Hours : 03

IA Marks : 25

Total No. of Lecture Hours : 50

No. of Lecture Hours/Week : 04 Exam Marks : 80

SQL The Relational Database Standard

4.1 Data Definition, Constraints, and Schema Changes in SQL2

4.2 Basic Queries in SQL

4.3 More Complex SQL Queries

 Page 58

DATABASE MANAGEMENT SYSTEM (21CS53)



**MODULE 2 SQL The Relational Database Standard 4.1 Data Definition, Constraints, and Schema Changes in SQL2**

• *Structured Query Language (SQL)* was designed and implemented at IBM Research. • Created in late 70‘s, under the name of SEQUEL

• A standard version of SQL (ANSI 1986), is called SQL86 or SQL1.

• A revised version of standard SQL, called SQL2 (or SQL92).

• SQL are going to be extended with objectoriented and other recent database concepts. • Consists of

• A Data Definition Language (DDL) for declaring database schemas

• Data Manipulation Language (DML) for modifying and querying database instances

• In SQL, relation, tuple, and attribute are called *table, row*, and *columns* respectively. • The SQL commands for data definition are *CREATE, ALTER*, and *DROP*. • The *CREATE TABLE* Command is used to specify a new table by giving it a name and specifying its attributes (columns) and constraints.

• Data types available for attributes are:

o *Numeric* integer, real (formated, such as *DECIMAL(10,2)*)

o *CharacterString* fixedlength and varyinglength

o *BitString* fixedlength, varyinglength

o *Date* in the form YYYYMMDD

o *Time* in the form HH:MM:SS

o *Timestamp* includes both the *DATE* and *TIME* fields

o *Interval* to increase/decrease the value of date, time, or timestamp

**4.2 Basic Queries in SQL**

• SQL allows a table (relation) to have two or more tuples that are identical in all their attributes values. Hence, an **SQL table** is not a set of tuple, because a set does not allow two identical members; rather it is a multiset of tuples.

• A basic query statement in SQL is the *SELECT* statement.

• The *SELECT* statement used in SQL has no relationship to the *SELECT* operation of relational algebra.

**The SELECT Statement**

The syntax of this command is:

**SELECT <attribute list>**

**FROM <table list>**

**WHERE <Condition>;**

Some example:

Page 59

DATABASE MANAGEMENT SYSTEM (21CS53)



Query 0: Retrieve the birthday and address of the employee(s) whose name is ‗John B. Smith‘ Q0: SELECT BDATE, ADDRESS

FROM EMPLOYEE

WHERE FNAME = ‗John‘ AND MINIT =‗B‘ AND LNAME = ‗SMITH‘ Query 1: Retrieve the name and address of all employee who work for the ‗Research‘ Dept. Q1: SELECT FNAME, LNAME, ADDRESS

FROM EMPLOYEE, DEPARTMENT

WHERE DNAME = ‗Research‘ AND DNUMBER = DNO

Query 2: For every project located in ‗Stafford‘, list the project number, the controlling department number, and the department manager‘s last name, address, and birthdate.

Q2: SELECT PNUMBER, DNUM, LNAME, ADDRESS,

BDATE FROM PROJECT, DEPARTMENT, EMPLOYEE

WHERE DNUM=DNUMBER AND MGRSSN=SSN AND PLOCATION = ‗Stafford‘

**Dealing with Ambiguous Attribute Names and Renaming (Aliening)**

Ambiguity in the case where attributes are same name need to qualify the attribute using DOT separator

e.g., WHERE DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER

More

Ambiguity in the case of queries that refer to the same relation twice

Query 8: For each employee, retrieve the employee‘s first and last name and the first and last name of his or her immediate supervisor

Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME FROM EMPLOYEE AS E, EMPLOYE

DATABASE MANAGEMENT SYSTEM (21CS53)



WHERE E.SUPERSSN=S.SSN

**Unspecified WHEREClause and Use of Asterisk (\*)**

A missing WHEREclause indicates no conditions, which means all tuples are selected In case of two or more table, then all possible tuple combinations are selected

Example: Q10: Select all EMPLOYEE SSNs , and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME

SELECT SSN, DNAME

FROM EMPLOYEE, DEPARTMENT

More

To retrieve all the attributes, use \* in SELECT clause

Retrieve all employees working for Dept. 5

SELECT \*

FROM EMPLOYEE

WHERE DNO=5

**Substring Comparisons, Arithmetic Operations, and Ordering**

• like, binary operator for comparing strings

• %, wild card for strings

• \_, wild card for characters

• ||, concatenate operation for strings

(name like ‘%a\_‘) is true for all names having ‗a‘ as second letter from the end.

• Partial strings are specified by using '

• SELECT FNAME, LNAME

• FROM EMPLOYEE

WHERE FNAME LIKE '%Mc%';

• In order to list all employee who were born during 1960s we have the followings: • SELECT FNAME, LNAME

. FROM EMPLOYEE

WHERE BDATE LIKE '6\_\_\_\_\_\_\_';

Page 61

DATABASE MANAGEMENT SYSTEM (21CS53)



• SQL also supports addition, subtraction, multiplication and division (denoted by +, , \*, and /, respectively) on numeric values or attributes with numeric domains.

**Examples:** Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

SELECT FNAME, LNAME, 1.1\*SALARY

FROM EMPLOYEE, WORKS\_ON, PROJECT

WHERE SSN=ESSN AND PNO=PNUMBER AND PNAME='ProductX';

Retrieve all employees in department number 5 whose salary between $30000 and $40000.

SELECT \*

FROM EMPLOYEE

WHERE (SALARY BETWEEN 30000 AND 40000) AND DNO=5;

It is possible to order the tuples in the result of a query.

SELECT DNAME, LNAME, FNAME, PNAME

FROM DEPARTMENT, EMPLOYEE, WORKS\_ON, PROJECT

WHERE DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER

ORDER BY DNAME, LNAME, FNAME;

The default order is in ascending order, but user can specify

ORDER BY DNAME DESC, LNAME ASC, FNAME, ASC;

**Tables as Sets in SQL**

SQL treats table as a **multiset,** which means duplicate tuples are OK

SQL does not delete duplicate because Duplicate elimination is an expensive operation (sort and delete) user may be interested in the result of a query in case of aggregate function, we do not want to eliminate duplicates

To eliminate duplicate, use DISTINCT

examples

Q11: Retrieve the salary of every employee , and (Q!2) all distinct salary values Q11: SELECT ALL SALARY

FROM EMPLOYEE

Q12: SELECT DISTINCT SALARY



Page 62

DATABASE MANAGEMENT SYSTEM (21CS53)



FROM EMPLOYEE

**4.3 More Complex SQL Queries**

Complex SQL queries can be formulated by composing nested SELECT/FROM/WHERE clauses within the WHEREclause of another query

Example: Q4: Make a list of Project numbers for projects that involve an employee whose last name is ‗Smith‘, either as a worker or as a manger of the department that controls the project

Q4 SELECT DISTINCT PNUMBER

FROM PROJECT

WHERE PNUMBER IN (SELECT PNUMBER

FROM PROJECT, DEPARTMENT, EMPLOYEE

WHERE DNUM=DNUMBER AND MGRSSN=SSN AND

LNAME=‗Smith‘

OR PNUMBER IN (SELECT PNO

FROM WORKS\_ON, EMPLOYEE

WHERE ESSN=SSN AND LNAME=‗Smith‘)

**IN operator and set of unioncompatible tuples**

Example:

SELECT DISTINCT ESSN

FROM WORKS\_ON

WHERE (PNO, HOURS) IN (SELECT PNO, HOURS

FROM WORKS\_ON

WHERE SSN=‗123456789‘

ANY, SOME and >, <=,<>,etc.



Page 63

DATABASE MANAGEMENT SYSTEM (21CS53)



**The keyword ALL**

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v to a set of multiset V.

ALL V returns TRUE if v is greater than all the value in the set

Select the name of employees whose salary is greater than the salary of all the employees in department 5

SELECT LNAME, FNAME

FROM EMPLOYEE

WHERE SALARY > ALL (SELECT SALARY

FROM EMPLOYEE

WHERE DNO=5);

**Ambiguity in nested query**

SELECT E.FNAME, E.LNAME

FROM EMPLOYEE AS E

WHERE E.SSN IN (SELECT ESSN

FROM DEPENDENT

WHERE ESSN=E.SSN AND E.FNAM=DEPENDENT\_NAME AND SEX=E.SEX

**Correlated Nested Query**

Whenever a condition in the WHEREclause of a nested query references some attributes of a relation declared in the outer query, the two queries are said to be correlated. The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) the outer query.

In general, any nested query involving the = or comparison operator IN can always be rewritten as a single block query

SELECT E.FNAME, E.LNAME

FROM EMPLOYEE E, DEPENDENT D

WHERE E.SSN=D.ESSN AND E.SEX=D.SEX AND E.FNAME =D.DEPENDENT=NAME

Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.



Page 64

DATABASE MANAGEMENT SYSTEM (21CS53)



Q12: SELECT E.FNAME, E.LNAME

FROM EMPLOYEE AS E

WHERE E.SSN IN (SELECT ESSN

FROM DEPENDENT

WHERE ESSN=E.SSN AND

E.FNAME=DEPENDENT\_NAME)

In Q12, the nested query has a different result for each tuple in the outer query.

The original SQL as specified for SYSTEM R also had a CONTAINS comparison operator, which is used in conjunction with nested correlated queries This operator was dropped from the language, possibly because of the difficulty in implementing it efficiently Most implementations of SQL do not have this operator The CONTAINS operator compares two sets of values , and returns TRUE if one set contains all values in the other set (reminiscent of the division operation of algebra).

Query 3: Retrieve the name of each employee who works on all the projects controlled by department number 5.

Q3: SELECT FNAME, LNAME

FROM EMPLOYEE WHERE ( (SELECT PNO FROM WORKS\_ON WHERE SSN=ESSN) CONTAINS (SELECT PNUMBER FROM PROJECT WHERE DNUM=5) )

In Q3, the second nested query, which is not correlated with the outer query, retrieves the project numbers of all projects controlled by department 5.

The first nested query, which is correlated, retrieves the project numbers on which the employee works, which is different for each employee tuple because of the correlation.

**THE EXISTS AND UNIQUE FUNCTIONS IN SQL**

EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not We can formulate Query 12 in an alternative form that uses EXISTS as Q12B below.

Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

SELECT E.FNAME, E.LNAME

FROM EMPLOYEE E

WHERE EXISTS (SELECT \*

FROM DEPENDENT



Page 65

DATABASE MANAGEMENT SYSTEM (21CS53)



WHERE E.SSN=ESSN AND SEX=E.SEX AND

E.FNAME=DEPENDENT\_NAME

Query 6: Retrieve the names of employees who have no dependents.

Q6: SELECT FNAME, LNAME

FROM EMPLOYEE

WHERE NOT EXISTS (SELECT \*

FROM DEPENDENT

WHERE SSN=ESSN)

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If none exist , the EMPLOYEE tuple is selected EXISTS is necessary for the expressive power of SQL

**EXPLICIT SETS AND NULLS IN SQL**

It is also possible to use an explicit (enumerated) set of values in the WHEREclause rather than a nested query Query 13: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

Retrieve SSNs of all employees who work on project number 1,2,3

SELECT DISTINCT ESSN

FROM WORKS\_ON

WHERE PNO IN (1,2,3)

**Null example**

SQL allows queries that check if a value is NULL (missing or undefined or not applicable) SQL uses IS or IS NOT to compare NULLs because it considers each NULL value distinct from other NULL values, so equality comparison is not appropriate .

Retrieve the names of all employees who do not have supervisors

SELECT FNAME, LNAME

FROM EMPLOYEE

WHERE SUPERSSN IS NULL

Note: If a join condition is specified, tuples with NULL values for the join attributes are not included in the result

**Join Revisit**

****

****Mr. MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 66

DATABASE MANAGEMENT SYSTEM (21CS53)



Retrieve the name and address of every employee who works for ‗Search‘ department

SELECT FNAME, LNAME, ADDRESS

FROM (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)

WHERE DNAME=‗Search‘

**Aggregate Functions**

Include COUNT, SUM, MAX, MIN, and AVG

Query 15: Find the sum of the salaries of all employees the ‗Research‘ dept, and the max salary, the min salary, and average:

SELECT SUM(SALARY), MAX(SALARY), MIN(SALARY) AVG(SALARY) FROM EMPLOYEE

WHERE DNO=FNUMBER AND DNAME=‗RSEARCH‘

Query 16: Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

Q16: SELECT MAX(SALARY), MIN(SALARY), AVG(SALARY)

FROM EMPLOYEE, DEPARTMENT

WHERE DNO=DNUMBER AND DNAME='Research'

Queries 17 and 18: Retrieve the total number of employees in the company (Q17), and the number of employees in the 'Research' department (Q18).

Q17: SELECT COUNT (\*)

FROM EMPLOYEE

Q18: SELECT COUNT (\*)

FROM EMPLOYEE, DEPARTMENT

WHERE DNO=DNUMBER AND DNAME='Research'

**Example of grouping**

In many cases, we want to apply the aggregate functions to subgroups of tuples in a relation Each subgroup of tuples consists of the set of tuples that have the same value for the grouping attribute(s)

The function is applied to each subgroup independently

 Page 67

DATABASE MANAGEMENT SYSTEM (21CS53)



SQL has a GROUP BYclause for specifying the grouping attributes, which must also appear in the SELECTclause

For each project, select the project number, the project name, and the number of employees who work on that projet

SELECT PNUMBER, PNAME, COUNT(\*)

FROM PROJECT, WORKS\_ON

WHERE PNUMBER=PNO

GROUP BY PNUMBER, PNAME

In Q20, the EMPLOYEE tuples are divided into groupseach group having the same value for the grouping attribute DNO The COUNT and AVG functions are applied to each such group of tuples separately.The SELECTclause includes only the grouping attribute and the functions to be applied on each group of tuples. A join condition can be used in conjunction with grouping

Query 21: For each project, retrieve the project number, project name, and the number of employees who work on that project.

Q21: SELECT PNUMBER, PNAME, COUNT (\*)

FROM PROJECT, WORKS\_ON

WHERE PNUMBER=PNO

GROUP BY PNUMBER, PNAME

In this case, the grouping and functions are applied after the joining of the two relations

THE HAVINGCLAUSE:

Sometimes we want to retrieve the values of these functions for only those groups that satisfy certain conditions. The HAVINGclause is used for specifying a selection condition on groups (rather than on individual tuples)

Query 22: For each project on which more than two employees work , retrieve the project number, project name, and the number of employees who work on that project.

Q22: SELECT PNUMBER, PNAME, COUNT (\*)

FROM PROJECT, WORKS\_ON

WHERE PNUMBER=PNO

GROUP BY PNUMBER, PNAME

HAVING COUNT (\*) > 2



Page 68

DATABASE MANAGEMENT SYSTEM (21CS53)



**Questions**

1. Explain how groupBy clause works?What is the difference between WHERE and Having? 2. How does SQL inplement the entity integrity constraints of relational Data Model?Explain with an example?

3. With respect to SQL, explain with example.

4. Explain IN and EXISTS operations with an example

5. Explain UNIQUE and INTERSECTION operations with an example

Page 69

DATABASE MANAGEMENT SYSTEM (21CS53)



**MODULE 3**

**SQL 2 The Relational Database Standard**

Subject Code : **21CS53** Exam Hours : 03

IA Marks : 25

Total No. of Lecture Hours : 50

No. of Lecture Hours/Week : 04 Exam Marks : 80

SQL 2 The Relational Database Standard

5.1 Update Statements in SQL

5.2 Views in SQL

5.3 Additional features

5.4 Database Programming

5.5 Embedded SQL

5.6 Dynamic SQL

5.7 Database stored procedures and SQL/PSM

Page 70

DATABASE MANAGEMENT SYSTEM (21CS53)



**MODULE 3 SQL The Relational Database Standard 5.1 Update Statements in SQL**

**The Insert Command**

INSERT INTO EMPLOYEE

VALUES (‗Richard‘,‘K‘,‘Marini‘,653298653‘,‘30dec52‘,98 Oak Forest, Katy, TX‘,‘M‘,37000,‘987654321‘,4)

More on Insert

Use explicit attribute names:

INSERT INTO EMPLOYEE (FNAME, LNAME,SSN)

VALUES (‗Richard‘,‘Marini‘, ‗653298653‘

**The DELECT Command**

DELETE FROM EMPLOYEE

WHERE LNAME=‗Brown‘

**The UPDATE Command**

Used to modify values of one or more selected tuples

Change the location and controlling department number of project number 10 to ‗Bellaire‘ and 5 respectively

UPDATE PROJECT

SET PLOCATION = ‗Bellaire‘, DNUM=5

Where PNUMBER=10;

**5.2 Views in SQL**

A view refers to a single table that is derived from other tables

CREATE VIEW WORKS\_ON1



Mr. MOHAMMED ZIAULLA, Assistant Professor Dept of CSE, KNSIT Page 71

DATABASE MANAGEMENT SYSTEM (21CS53)



AS SELECT FNAME, LNAME, PNAME, HOURS

FROM EMPLOYEE, PROJECT, WORKS\_ON WHERE SSN=ESSN AND PNO=PNUMBER More on View

CREATE VIEW DEPT\_INFO(DEPT\_NAME, NO\_OF\_EMPLS, TOTAL\_SAL) AS SELECT DNAME, COUNT(\*), SUM(SALARY)

FROM DEPARTMENT, EMPLOYEE

WHERE DNUMBER=DNO

GROUP BY DNAME

More on view

Treat WORKS\_ON1 like a base table as follows

SELECT FNAME, LNAME

FROM WORKS\_ON1

WHERE PNMAE=‗PROJECTX‘

**Main advantage of view:**

Simplify the specification of commonly used queries

More on View

A View is always up to date;

A view is realized at the time we specify(or execute) a query on the view

DROP VIEW WORKS\_ON1

**Updating of Views**

Updating the views can be complicated and ambiguous

In general, an update on a view on defined on a single table w/o any aggregate functions can be mapped to an update on the base table



Page 72

DATABASE MANAGEMENT SYSTEM (21CS53)



More on Views

We can make the following observations:

A view with a single defining table is updatable if we view contain PK or CK of the base table View on multiple tables using joins are not updatable

View defined using grouping/aggregate are not updatable

Specifying General Constraints

Users can specify certain constraints such as semantics constraints

CREATE ASSERTION SALARY\_CONSTRAINT

CHECK ( NOT EXISTS ( SELECT \* FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D

WHERE E.SALARY > M. SALARY **AND** E.DNO=D.NUMBER **AND** D.MGRSSN=M.SSN)) **5.3 Additional features**

Granting and revoking privileges

Embedding SQL statements in a general purpose languages (C, C++, COBOL, PASCAL)

SQL can also be used in conjunction with a general purpose programming language, such as PASCAL, COBOL, or PL/I. The programming language is called the host language. The embedded SQL statement is distinguished from programming language statements by prefixing it with a special character or command so that a preprocessor can extract the SQL statements. In PL/I the keywords EXEC SQL precede any SQL statement. In some implementations, SQL statements are passed as parameters in procedure calls. We will use PASCAL as the host programming language, and a "$" sign to identify SQL statements in the program. Within an embedded SQL command, we may refer to program variables, which are prefixed by a "%" sign. The programmer should declare program variables to match the data types of the database attributes that the program will process.These program variables may or may not have names that are identical to their corresponding attributes.

Example: Write a program segment (loop) that reads a social security number and prints out some information from the corresponding EMPLOYEE tuple

E1: LOOP:= 'Y';

while LOOP = 'Y' do

begin

writeln('input social security number:');

Page 73

DATABASE MANAGEMENT SYSTEM (21CS53)



readln(SOC\_SEC\_NUM);

$SELECT FNAME, MINIT, LNAME, SSN,

BDATE, ADDRESS, SALARY

INTO %E.FNAME, %E.MINIT, %E.LNAME, %E.SSN,

%E.BDATE, %E.ADDRESS, %E.SALARY

FROM EMPLOYEE

WHERE SSN=%SOC\_SEC\_NUM ;

writeln( E.FNAME, E.MINIT, E.LNAME,

E.SSN, E.BDATE, E.ADDRESS, E.SALARY);

writeln('more social security numbers (Y or N)? ');

readln(LOOP)

end;

In E1, a single tuple is selected by the embedded SQL query; that is why we are able to assign its attribute values directly to program variables. In general, an SQL query can retrieve many tuples. The concept of a cursor is used to allow tupleatatime processing by the PASCAL programCURSORS: We can think of a cursor as a pointer that points to a single tuple (row) from the result of a query.The cursor is declared when the SQL query command is specified. A subsequent OPEN cursor command fetches the query result and sets the cursor to a position before the first row in the result of the query; this becomes the current row for the cursor. Subsequent FETCH commands in the program advance the cursor to the next row and copy its attribute values into PASCAL program variables specified in the FETCH command. An implicit variable SQLCODE communicates to the program the status of SQL embedded commands. An SQLCODE of 0 (zero) indicates successful execution. Different codes are returned to indicate exceptions and errors. A special END\_OF\_CURSOR code is used to terminate a loop over the tuples in a query result. A CLOSE cursor command is issued to indicate that we are done with

the result of the query When a cursor is defined for rows that are to be updated the clause FOR UPDATE OF must be in the cursor declaration, and a list of the names of any attributes that will be updated follows.The condition WHERE CURRENT OF cursor specifies that the current tuple is the one to be updated (or deleted)

Example: Write a program segment that reads (inputs) a department name, then lists the names of employees who work in that department, one at a time. The program reads a raise amount for each employee and updates the employee's salary by that amount.

E2: writeln('enter the department name:'); readln(DNAME);

$SELECT DNUMBER INTO %DNUMBER

FROM DEPARTMENT

WHERE DNAME=%DNAME;

$DECLARE EMP CURSOR FOR

SELECT SSN, FNAME, MINIT, LNAME, SALARY

FROM EMPLOYEE

WHERE DNO=%DNUMBER

FOR UPDATE OF SALARY;

$OPEN EMP;

$FETCH EMP INTO %E.SSN, %E.FNAME, %E.MINIT,

%E.LNAME, %E.SA

Page 74

DATABASE MANAGEMENT SYSTEM (21CS53)



while SQLCODE = 0 do

begin

writeln('employee name: ', E.FNAME, E.MINIT, E.LNAME);

writeln('enter raise amount: '); readln(RAISE);

$UPDATE EMPLOYEE SET SALARY = SALARY + %RAISE

WHERE CURRENT OF EMP;

$FETCH EMP INTO %E.SSN, %E.FNAME, %E.MINIT,

%E.LNAME, %E.SAL;

end;

$CLOSE CURSOR EMP;

**5.4 Database Programming**

◼Objective:

◼To access a database from an application program (as opposed to interactive interfaces) ◼Why?◼An interactive interface is convenient but not sufficient

◼A majority of database operations are made thru application programs (increasingly  thru web applications)

◼Embedded commands:

◼Database commands are embedded in a general-purpose programming language

◼Library of database functions:

◼Available to the host language for database calls; known as an *API*

◼*API* standards for Application Program Interface

◼A brand new, full-fledged language

◼Minimizes impedance mismatch

Impedance Mismatch

◼Incompatibilities between a host programming language and the database model, e.g., ◼type mismatch and incompatibilities; requires a new binding for each language

◼set vs. record-at-a-time processing

◼need special iterators to loop over query results and manipulate individual values ◼Client program *opens a connection* to the database server

◼Client program *submits queries to and/or updates* the database

◼When database access is no longer needed, client program *closes (terminates) the connection* **5.5 Embedded SQL**

◼Most SQL statements can be embedded in a general-purpose *host* programming language such as  COBOL, C, Java



 Page 75

DATABASE MANAGEMENT SYSTEM (21CS53)



◼An embedded SQL statement is distinguished from the host language statements by  enclosing it between EXEC SQL or EXEC SQL BEGIN and a matching END-EXEC or EXEC SQL END (or semicolon)

◼Syntax may vary with language

◼*Shared variables* (used in both languages) usually prefixed with a colon (:) in

SQL

◼Variables inside DECLARE are shared and can appear (while prefixed by a colon) in SQL  statements

◼SQLCODE is used to communicate errors/exceptions between the database and the program int loop;

EXEC SQL BEGIN DECLARE SECTION;

varchar dname[16], fname[16], …;

char ssn[10], bdate[11], …;

int dno, dnumber, SQLCODE, …;

EXEC SQL END DECLARE SECTION;

◼Connection (multiple connections are possible but only one is

active) CONNECT TO server-name AS connection-name

AUTHORIZATION user-account-info;

◼Change from an active connection to another

one SET CONNECTION connection-name;

◼Disconnection

DISCONNECT connection-name;

loop = 1;

while (loop) {

prompt (―Enter SSN: ―, ssn);

EXEC SQL



Page 76

DATABASE MANAGEMENT SYSTEM (21CS53)



select FNAME, LNAME, ADDRESS, SALARY

into :fname, :lname, :address, :salary

from EMPLOYEE where SSN == :ssn;

if (SQLCODE == 0) printf(fname, …);

else printf(―SSN does not exist: ―, ssn);

prompt(―More SSN? (1=yes, 0=no): ―, loop);

END-EXEC

◼}A cursor (iterator) is needed to process multiple tuples

◼FETCH commands move the cursor to the *next* tuple

◼CLOSE CURSOR indicates that the processing of query results has been completed

◼Objective:

**5.6 Dynamic SQL**

Composing and executing new (not previously compiled) SQL statements at run-time ◼a program accepts SQL statements from the keyboard at run-time

◼a point-and-click operation translates to certain SQL query

◼Dynamic update is relatively simple; dynamic query can be complex

◼because the type and number of retrieved attributes are unknown at compile time

EXEC SQL BEGIN DECLARE SECTION;

varchar sqlupdatestring[256];

EXEC SQL END DECLARE SECTION;

…prompt (―Enter update command:―, sqlupdatestring);

EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring;

◼EXEC SQLSQLJ: a standard for embedding SQL in Java

◼An SQLJ translator converts SQL statements into Java

◼These are executed thru the *JDBC* interface

◼Certain classes have to be imported

◼E.g., java.sql



Page 77

DATABASE MANAGEMENT SYSTEM (21CS53)



EXECUTE sqlcommand;

◼*Environment record*:

◼Keeps track of database connections

◼*Connection record*:

◼Keep tracks of info needed for a particular connection

◼*Statement record*:

◼Keeps track of info needed for one SQL statement

◼*Description record*:

◼Keeps track of tuples

◼Load SQL/CLI libraries

◼Declare record handle variables for the above components (called: SQLHSTMT, SQLHDBC,  SQLHENV, SQLHDEC)

◼Set up an environment record using SQLAllocHandle

◼Set up a connection record using SQLAllocHandle

◼Set up a statement record using SQLAllocHandle

◼Prepare a statement using SQL/CLI function SQLPrepare

◼Bound parameters to program variables

◼Execute SQL statement via SQLExecute

◼Bound query columns to a C variable via SQLBindCol

◼Use SQLFetch to retrieve column values into C variables

**5.7 Database stored procedures and SQL/PSM**

◼Persistent procedures/functions (modules) are stored locally and executed by the database  server◼As opposed to execution by clients

◼Advantages:

◼If the procedure is needed by many applications, it can be invoked by any of them (thus  reduce duplications)

◼Execution by the server reduces communication costs

◼Enhance the modeling power of views

◼Disadvantages:

◼Every DBMS has its own syntax and this can make the system less portable

◼A stored procedure

CREATE PROCEDURE procedure-name (params)

local-declarations

procedure-body;

◼A stored function



Page 78