BHARATH COLLEGE OF SCIENCE AND MANAGEMENT, THANJAVUR.

B.Sc INFORMATION TECHNOLOGY –VI SEMESTER

(For the candidates admitted from the academic year 2016-2017 onwards)

CORE COURSE IX - DATA BASE SYSTEMS

Unit I

Introduction: Database-System Applications- Purpose of Database Systems - View of Data -- Database Languages - Relational Databases - Database Design -Object-Based and Semi structured Databases - Data Storage and Querying Transaction Management -Data Mining and Analysis - Database Architecture - Database Users and Administrators - History of Database Systems.

Unit II

Relational Model: Structure of Relational Databases - Fundamental Relational-Algebra Operations Additional Relational-Algebra Operations- Extended Relational-Algebra Operations - Null Values - Modification of the Database.

Unit III

SQL: Data Definition - Basic Structure of SQL Queries - Set Operations-Aggregate Functions - Null Values- Nested Sub queries - Complex Queries - Views -Modification of the Database - Joined Relations - SQL Data Types and Schemas - Integrity Constraints -Authorization - Embedded SQL

Unit IV

Relational Languages: The Tuple Relational Calculus - The Domain Relational Calculus - Query-by- Example. Database Design and the E-R Model: Overview of the Design Process - The Entity-Relationship Model - 3 Constraints - Entity-Relationship Diagrams - Entity-Relationship Design Issues - Weak Entity Sets - Database Design for Banking Enterprise

Unit V

Relational Database Design: Features of Good Relational Designs - Atomic Domains and First Normal Form - Decomposition Using Functional Dependencies - Functional-Dependency Theory - Decomposition Using Functional Dependencies - Decomposition Using Multivalve Dependencies-More Normal Forms - Database-Design Process

Text Book:

1. Database System Concepts, Fifth edition, Abraham Silberschatz , Henry F. Korth, S. Sudarshan, McGraw-Hill-2005.

Reference Books:

1. “An introduction to database systems”, Bipin C. Desai, Galgotia Publications Pvt Ltd, 1991.

2. “An Introduction to Database Systems”, C.J.Date, Third Edition Addison Wesl

Unit I

2 Marks

1. What is database?

- A database is a collection of information that is organized so that it can easily be accessed, managed, and updated.
- Some examples of database software are Oracle, FileMaker Pro, Microsoft Access, Microsoft SQL Server, SAP and MySQL
- Databasesoftware, also called a database management system or DBMS, is used to store, retrieve, add, delete and modify data.

2. What is Database Management System (DBMS)?

- A database management system (DBMS) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data
- A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases
- DBMS contains information about a particular enterprise
    1. Collection of interrelated data
    2. Set of programs to access the data
    3. An environment that is both *convenient* and *efficient* to use.

3. What are the advantages of DBMS?

- Redundancy is controlled
- Unauthorized access restricted.
- Providing multiple user interface
- Enforcing integrity constraints.
- Providing backup and recovery.

4. List out the disadvantages in File Processing System.

- Data redundancy & inconsistency
- Difficult in accessing data
- Data isolation
- Data integrity
- Concurrent access is not possible
- Security Problems.

5. What are the major disadvantages of a database system?

- Setup of the database system requires much more knowledge, money, skills, and time
- Complexity of the database may result in poor performance

6. List the different applications of DBMS.

- Banking: all transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Credit card transactions
- Tele communications
- Finance
- Sales: customers, products, purchases
- Online retailers: order tracking, customized recommendations
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions

7. Define Data independence.

- The ability to modify a schema definition in only level without affecting a Schema definition in the next higher level is called data independence.

8. Give the levels of data abstraction?

- Physical level
- Logical level
- View level

9. Write about the instances and Schemas

- The collection of information stored in the database at a particular moment is called an instance of the database.
- The overall design of the database is called the database schema. Schemas are changed infrequently
- Schema represents the variable declaration in a program.
- The value of the variable in a program is an instant of a database schema.
- A database may also have several schemas at the view level and is called as sub-schemas

10. Define Storage Manager.

- Storage manger is a program module that provides the interface between the low levels data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
    - Interaction with the file manager
    - Translation of DML commands in to low level file system commands
    - Efficient storing, retrieving and updating of data.

11. What are the components of storage manager?

- The storage manager components include
    - Authorization and integrity manager
    - Transaction manager
    - File manager
    - Buffer manager

12. What are the data structures implemented by Storage manager?

- Data files – which store the database itself
- Data dictionary – which stores metadata about the structure of the database in particular the schema of the database
- Indices – which can provide fast access to data items

13. Define data model and list out their types.

- A collection of conceptual tools for describing Data, Data relationships, Data semantics, consistency constraints.
- Types of data Model
    - Relational model
    - Entity Relationship data model (mainly for database design)
    - Object based data models (Object oriented and Object relational)
    - Semi structured data model (XML)
    - Other older models:
    - Network model
    - Hierarchical model

14. What is E-R model?

- The entity-relationship data model is based on perception of a real world that consists of a collection of basic objects, called entities and of relationships among these object

15. Define entity and entity set.

- An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person in an entity.
- The set of all entities of the same type are termed and entity set.

16. Define relationship and relationship set.

- A relationship is an association among several entities. For example, a depositor relationship associated a customer with each account.
- The set of all relationships of the same type are termed a relationship set.

17. What is Object Oriented Model?

- The model is based on collection of object.
- An object contains values stored in instance variables within the object.
- An object also contains bodies of code that operate on the object. These bodies of code are called methods.
- Objects that contain same type of values and the same methods are grouped together into classless

18. Define Record-Based Logical Models.

- Record-based logical models are used in describing data at the logical and levels.
- They are used both to specify the overall structure of the database and provide a high-level description of the implementation.

19. Define Relational model.

- The relational model uses a collection of tables to represent both data and the relationships among those data.
- Each table has multiple columns, and each column has a unique name.

20. Define Network model.

- Data in the networks model are represented by collection of records and relationships among data are represented by links, which can be viewed as pointers.
- The records in the database are organized as collections of arbitrary graphs.

21. Define Hierarchical Model

- The hierarchical model is similar to the network models in the sense that data and relationship among data are represented by records and links respectively.

- It differ from the network model in that the records are organized as collection of trees rather that arbitrary graphs.

22. Define Semi-structured data model

- The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes.
- Every data item of a particular type must have the same set of attributes.
- The Extensible Markup Language (XML) is widely used to represent semi-structured data

23. What is a data dictionary?

- A data dictionary is a data structure which stores Meta data about the structure of the database that is the schema of the database.

24. What are the Database languages?

- Database languages are used for read, update and store data in adatabase.
- There are several such languages that can be used for this purpose; one of them is SQL (Structured Query Language).
- Types of DBMS languages:
    - Data Definition Language (DDL) - to specify the database schema
    - Data Manipulation Language (DML) - to express database queries and updates.

25. What is a DML?

- A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model
- DML used to
    - Retrieval of information stored in the database
    - Insertion of new information into the database
    - Deletion of information from the database
    - Modification of information stored in the database

26. What are the types of DML?

- Procedural DMLs - require a user to specify what data are needed and how to get those data.
- Declarative DMLs or nonprocedural DMLs - require a user to specify what data are needed without specifying how to get those data.

27. What is a query and query language?

- A query is a statement requesting the retrieval of information.
- The portion of a DML that involves information retrieval is called a query language.

28. What is a DDL?

- DDL is used to specify a database schema by a set of definitions.
- DDL is also used to specify additional properties of the data.
- DDL is also called as data storage and definition language since it is used to specify the storage structure and access methods used by the database system.

29. What is an Integrity constraint?

- Integrity means something like 'be right' and consistent.
- The data in a database must be right and in good condition.
- Integrity constraints are
    - Domain constraint
    - Referential integrity
    - Assertion
    - Authorization

30. What is a Domain constraint?

- A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types)
- Domain constraints are the most elementary form of integrity constraint.
- They are tested easily by the system whenever a new data item is entered into the database

31. What is a Referential integrity?

- There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity).

32. What is an assertion?

- An assertion is any condition that the database must always satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions

33. What is an authorization?

- We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database.
- These differentiations are expressed in terms of authorization.
- Types of authorizations are
    - Read authorization
    - Insert authorization
    - Update authorization

- Delete authorization

34. What is normalization?

- In relational database design, the process of organizing data to minimize redundancy.
- Normalization usually involves dividing a database into two or more tables and defining relationships between the tables.

35. What are the query processors?

- DDL interpreter, which interprets DDL statements and records the definitions in the data dictionary.
- DML compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

36. What are the database users?

- Users are differentiated by the way they expect to interact with the system
  - **Application programmers**–interact with system through DML calls
  - **Sophisticated users**–form requests in a database query language
  - **Specialized users**–write specialized database applications that do not fit into the traditional data processing framework
  - **Naive users**–invoke one of the permanent application programs that have been written previously

37. What are the data base subsystems?

A database system has several subsystems.

- The **storage manager** subsystem provides the interface between the low level data stored in the database and the application programs and queries submitted to the system.
- The **query processor** subsystem compiles and executes DDL and DML statements.

<u>5 Marks</u>

38. List out the disadvantages of File processing System

- **Data redundancy and inconsistency**
  - **Data redundancy** means two different fields within a single database, or two different spots in multiple software environments or platforms.
  - **Data inconsistency** is a condition that occurs between files when similar **data** is kept in different formats in two different files, or when matching of **data** must be done between files

- **Difficulty in accessing data**
  - Difficulty in accessing data arises whenever there is no application program for a specific task.
- **Data isolation:**
  - This problem arises due to the scattering of data in various files with various formats. Due to the above disadvantages of the earlier data processing system, the necessity for an effective data processing system arises.
- **Integrity problems**.
  - The data values stored in the database must satisfy certain types of consistency constraints
- **Atomicity Problems**
  - An **atomic** transaction is an indivisible and irreducible series of database operations such that either all occur, or nothing occurs. A guarantee of **atomicity** prevents updates to the database occurring only partially, which can cause greater **problems** than rejecting the whole series outright.
- **Concurrent access anomalies**
  - If multiple users are updating the same data simultaneously it will result in inconsistent data state
- **Security problems**
  - Not every user of the database system should be able to access all the data.

39. List out the DBMS applications in detail

- **Enterprise Information**
  - **Sales**: For customer, product, and purchase information.
  - **Accounting**: For payments, receipts, account balances, assets and other accounting information.
  - **Humanresources**: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
  - **Manufacturing**: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items
  - **Onlineretailers**: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

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

40. Discuss briefly about view of data.

- A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how data stored and maintained.

  Data abstraction
- For the system to be useful, it must retrieve data efficiently. The need for the efficiency has to lead designers to use complex data structures to represent data in a database.
- The three levels of Data abstraction
  - Physical level
  - Logical level
  - View level
- Physical level:
  - The lowest level of abstraction describes how the data are actually stored.
  - The physical level describes complex low-level data structures in detail.
- Logical level:
  - The next-higher level of abstraction
  - It describes data stored in database, and the relationships among the data.

```
type instructor = record
                    ID : char (5);
                    name : char (20);
                    dept_name : char (20);
                    salary : numeric (8,2);
                  end;
```

- View level:
  - The highest level of abstraction describes only part of the entire database
  - The application programs hide details of data types.
  - Views can also hide information (such as an employee's salary) for security purposes.

41. Explain about ACID in DBMS

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- **ACID** Properties
  - A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:
  - **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
  - **Consistency** Execution of a transaction in isolation preserves the consistency of the database.
  - **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

42. Explain about the Database Administrator (DBA) and the functions of DBA

- A person who has such central control over the system is called a database Administrator.
- Data coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.
- The functions of DBA include:
  - Schema definition - The DBA creates the original database schema by executing a set of data definition statements in the DDL.
  - Storage structure and access method definition

- Schema and physical organization modification
    - The DBA carries out changes to the schema and physical organization, or to alter the physical organization to improve performance.
- Granting user authority to access the database
    - By granting different type of authorization, the database administrator can regulates which parts of the database various users can access.
    - The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- Routine maintenances.
    - Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disaster such as flooding.
    - Monitoring performance and responding to changes in requirements
    - Ensuring that enough free disk space is available or normal operations, and upgrading disk space as required.

10 Marks

43. Explain the history of DBMS

- 20^{th} century - Punched Cards – invented by Herman Hollerith. Used to record US Census data and mechanical systems were used to process the cards and tabulate the results.
- 1950's and early 1960's
    - Magnetic tapes were developed for data storage
    - Data could also be input from punched card decks and output to printers
- Late 1960's and 1970's
    - Hard disks widely used for data processing
    - The position of the data on the disk was immaterial. Since any location on disk could be accessed in just tens of milliseconds
- 1970's – CODD invented landmark paper defined the relational model and non-procedural ways of querying data
- 1980's
    - Relations databases had become competitive with network and hierarchical database system even in the area of performance

- Researches on parallel and distributed database and also on object oriented databases

➢ Early 1990's

- SQL language was designed primarily for decision support applications
- Tools for analyzing large amounts of data saw large growths in usage
- Vendors used parallel database products in this period

➢ 1990's

- The major event of the 1990s the growth of WWW
- DB systems had to support very high transaction processing rates as well as very high reliability and 24 * 7 availability
- No downtime for scheduled maintenance activities.
- DB system also had to be support web interfaces to data.

➢ 2000's

- The first half of the 2000s saw the emerging of XML and the associated query language XQuery as a db technology
- Although XML is widely used for data exchange for storing certain complex data types

44. Discuss briefly about Data Models.

➢ A collection of tools for describing Data, Data relationships, Data semantics and Data constraints.

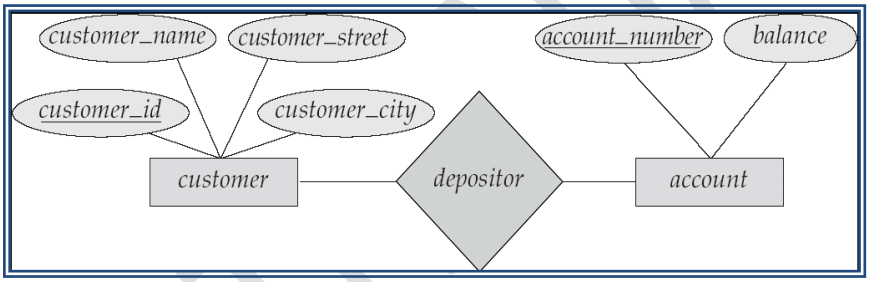
➢ **Relational model:**

- The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple column has a unique name.
- The relational model is an example of Record-based model
- The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

Example

customer_id	customer_name	customer_street	customer_city	account_number
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

- **The Entity-Relationship Model**
  - The entity-relationship (E-R) data Model is based on a perception of a real world that consists of a collection of *entities* and *relationships*
  - Entity: a "thing" or "object" in the enterprise that is distinguishable from other objects
    - Described by a set of *attributes*
    - Relationship: an association among several entities
- Represented diagrammatically by an entity-relationship diagram:



- **Object Relational Data Models**
  - Extend the relational data model by including object orientation and constructs to deal with added data types.
  - Allow attributes of tuples to have complex types, including non atomic values such as nested relations.
  - Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
  - Provide upward compatibility with existing relational languages.
- **Semi structured data model (XML)**
  - Defined by the WWW Consortium (W3C)
  - Originally intended as a document markup language not a database language
  - The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents

- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

45. Explain the Database Languages

- Database languages are used for read, update and store data in a database.
- There are several such languages that can be used for this purpose; one of them is SQL (Structured Query Language).
- Types of DBMS languages:
  - Data Definition Language (DDL) - to specify the database schema
  - Data Manipulation Language (DML) - to express database queries and updates
- DML
  - A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model
  - DML used to
    - Retrieval of information stored in the database
    - Insertion of new information into the database
    - Deletion of information from the database
    - Modification of information stored in the database
- DDL
  - DDL is used to specify a database schema by a set of definitions.
  - DDL is also used to specify additional properties of the data.
  - DDL is also called as data storage and definition language since it is used to specify the storage structure and access methods used by the database system
- DDL involved in consistency constraints
  - Domain Constraints
    - A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types)
  - Referential integrity
    - There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity).
  - Assertion

        - An assertion is any condition that the database must always satisfy.Domain constraints and referential-integrity constraints are special forms of assertions
    - Authorization
        - We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database.
        - These differentiations are expressed in terms of authorization.
        - Types of authorizations are
            - Read authorization
            - Insert authorization
            - Update authorization
            - Delete authorization

Unit II

2 Marks

1. **What is the relational model?**
   - The relational model is today the primary data model for commercial data processing applications
   - It attained its primary position because of its simplicity
2. **What is a relation, tuple, attribute?**
   - **Relation –** In relational model relation is used to refer to a table, each of which is assigned a unique name.
   - **Tuple –** Tuple is a row in the table
   - **Attribute –** refers to a column of a table.
   - **Relation instance –** to refer to a specific instance of a relation that is containing a specific set of rows.
   - **Domain -** For each attribute of a relation, there is a set of permitted values, called as domain.
   - **Atomic domain -** We require that for all relations r, the domains of all attributes of r be atomic. A domain is atomic if the elements of the domain are considered to be indivisible units.
3. **What are the different relational algebra operations?**
   - **Fundamental relational algebra operations:**
     a. Unary Operators - select, project and rename.
     b. Binary operator-product, set union and set difference
   - **Additional relational algebra**
     a. Set intersection, natural join, division and assignment
   - **Extended relational algebra**
     a. Outer join, aggregate and generalized projection
4. **Define keys.**
   - A way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can uniquely identify the tuple.
   - **Superkey** is a set of one or more attributes that, taken collectively allow us to identify uniquely a tuple in the relation.

- **Candidate keys** Which no proper subset is a superkey. Such minimal superkeys are called candidate keys.
- **Primary key:** a candidate key chosen as the principal means of identifying tuples within a relation
    a. Should choose an attribute whose value never, or very rarely, changes.

    E.g. email address is unique, but may change
- **Foreign key:** A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a foreign key.
    b. E.g. *customer_name* and *account_number* attributes of *depositor* are foreign keys to *customer* and *account* respectively.
    c. Only values occurring in the primary key attribute of the referenced relation may occur in the foreign key attribute of the referencing relation.

5. Define database schema and instance.

- Database schema is the logical design of the database.
- Database instance is a snapshot of the data in the database at a given instant in time.
- The concept of a relation schema corresponds to the programming language notion of type definition.

  E.g., Account_schema = (account_no, branch_name, balance)

  Branch_schema = (branch_name, branch_city, assets)

6. **What are Query languages?**

- A Query language is a language in which user requests information from the database. These languages are usually higher than that of standard programming language.
- **Categories of languages**
    - Procedural
    - Non-procedural, or declarative
- "**Pure" languages:**
    - Relational algebra
    - Tuple relational calculus
    - Domain relational calculus

  Pure languages form underlying basis of query languages that people use.

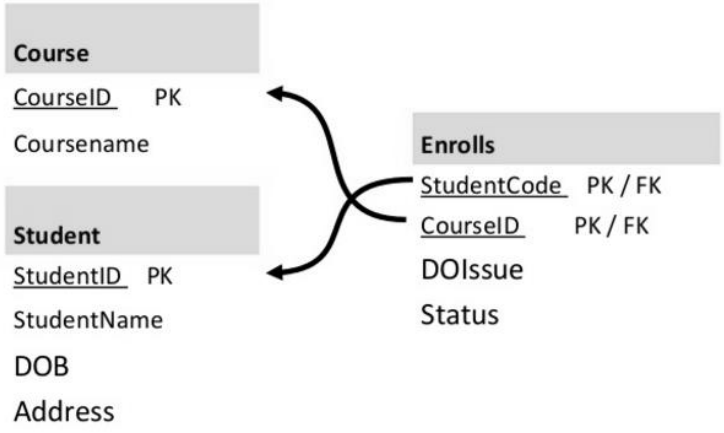
7. List out Additional relational algebra operation:

- Set intersection
- Natural join
- Outer Join
- Division

5 Marks

8. Explain the schema diagram

- A database schema is the skeleton structure that represents the logical view of the entire database.
- A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams.
- Below schema diagram shows the student database with Course, Student and Enrolls tables.
    i. Course table has CourseID as primary key field and Course name as normal field
    ii. Student table has StudentID as primary key field and StudentName, DOB, Address are the other fields
    iii. Enrolls table has the StudentCode field which referenced from Student's Student ID field and CourseID referenced from Course table.



9. Discuss in detail about fundamental relational algebra operations

Fundamental Operations

- The select, project, and rename operations are called *unary* operations, because they operate on one relation.
- The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

Symbol (Name)	Example of Use
σ	$\sigma_{salary>=85000}(instructor)$
(Selection)	Return rows of the input relation that satisfy the predicate.
Π	$\Pi_{ID,salary}(instructor)$
(Projection)	Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
⋈	$instructor \bowtie department$
(Natural join)	Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
×	$instructor \times department$
(Cartesian product)	Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
∪	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$
(Union)	Output the union of tuples from the two input relations.

- All procedural relational query languages provide a set of operations that can be applied to either a single relation or a pair of relations
- Selection of specific tuples from a single relation that satisfies some particular predicate for example salary >$85,000. This result in new relation that is subset of the original relation
- **Selection of specific attribute** (column) from the relation. The result in a new relation having only those selected attributes. For example dept_name from instructor table / relation.
- **The join operation** allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple. There are a number of different ways to join relations
- **The Cartesian product operation** combines tuples fromtworelations, but unlikethe join operation, its result contains all pairs of tuples from the two relations,regardless of whether their attribute values match.
- **The union operation** performs a set union of two "similarly structured" tables(say a table of all graduate students and a table of all undergraduate students).
- Relational algebra defines with a few of the operations below.

10 Marks

10. **Explain briefly about the basic structure of relational Model**

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

- **Basic Structure**
  - Formally, given sets D1, D2, …. Dn a **relation** r is a subset of

    D1x D2 x …x Dn

    Thus, a relation is a set of n-tuples (a1,a2, …, an) where each $a_i \in D_i$

Example: If

- customer_name= {Jones, Smith, Curry, Lindsay, …} /* Set of all customer names */
- customer_street= {Main, North, Park, …} /* set of all street names*/
- customer_city= {Harrison, Rye, Pittsfield, …} /* set of all city names */

Then r= { (Jones, Main, Harrison),

(Smith, North, Rye),

(Curry, North, Rye),

(Lindsay, Park, Pittsfield) }is a relation over

Customer_name x customer_street x customer_city

- **Attribute Types**
  - Each attribute of a relation has a name
  - The set of allowed values for each attribute is called the **domain** of the attribute
  - Attribute values are (normally) required to be **atomic**; that is, indivisible

    E.g. the value of an attribute can be an account number, but cannot be a set of account numbers
  - Domain is said to be atomic if all its members are atomic
  - The special value null is a member of every domain
  - The null value causes complications in the definition of many operations

    We shall ignore the effect of null values in our main presentation and consider their effect later
- **Relation Schema**
  - A1, A2, …, An are attributes
  - R= (A1, A2, …, An) is a relation schema

    Example:

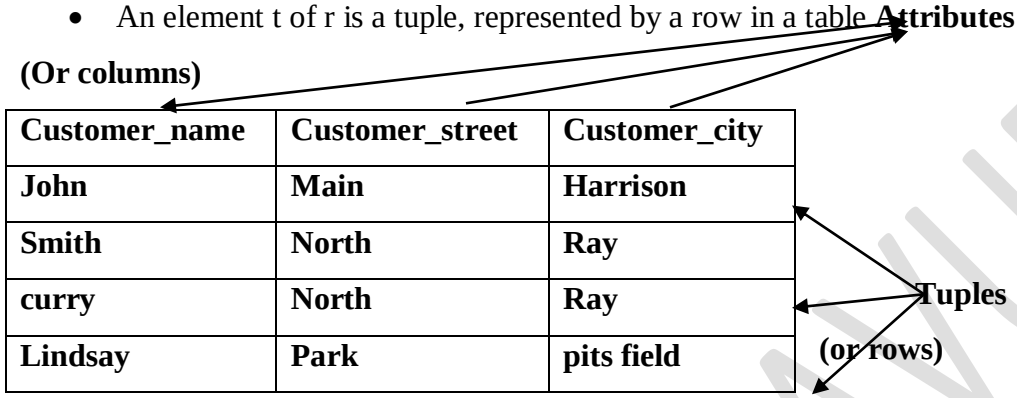
    Customer_schema= (customer_name, customer_street, customer_city)
  - r(R) denotes a relation on the relation schema R

    Example:

Customer (Customer_schema)

- **Relation Instance**
  - The current values (relation instance) of a relation are specified by a table
  - An element t of r is a tuple, represented by a row in a table **Attributes**

(Or columns)



Customer_name	Customer_street	Customer_city
John	Main	Harrison
Smith	North	Ray
curry	North	Ray
Lindsay	Park	pits field

Customer

- **Relations are Unordered**
  - Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
  - Example: account relation with unordered tuples

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

- **Database**
  - A database consists of multiple relations
  - Information about an enterprise is broken up into parts, with each relation storing one part of the information
    - account : stores information about accounts
    - depositor : stores information about which customer owns which account
    - customer : stores information about customers
  - Storing all information as a single relation such as
    bank (account_number, balance, customer_name, ..)
  - Results inrepetition of information
    e.g., if two customers own an account (What gets repeated?)

- The need for null valuese.g., to represent a customer without an account

11. Discuss briefly about the modification of the database.

- The content of the database may be modified using the following operations:
  a. Deletion
  b. Insertion
  c. Updating
- All these operations are expressed using the assignment operator.
- The customer Relation

Deletion:

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user; the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

  $r \leftarrow r - E$

  Where r is a relation and E is a relational algebra query.
- Delete all account records in the Perryridge branch.

  account ←account –σ branch_name = "Perryridge"(account)
- Delete all loan records with amount in the range of 0 to 50

  Loan ←loan–σamount ≥ 0 and amount ≤ 50(loan)
- Delete all accounts at branches located in Needham.

  r1←σbranch_city = "Needham"(account × branch)

  r2←∏account_number, branch_name, balance(r1)

  r3 ←∏customer_name, account_number(r2×depositor)

  account ←account –r2

  depositor ←depositor –r3

Insertion

- To insert data into a relation, we either:
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:

  r ←r∪E

Where r is a relation and E is a relational algebra expression.

- The insertion of a single tuple is expressed by letting E is a constant relation containing one tuple.

Insertion Examples

- Insert information in the database specifying that Smith has $1200 in account A-973 at the perryridge branch.

  account ←account∪{(“A-973”, “Perryridge”, 1200)}

  depositor ←depositor∪{(“Smith”, “A-973”)}

- Provide as a gift for all loan customers in the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

  r1←(σbranch_name = “Perryridge”(borrower loan))

  account ←account∪∏loan_number, branch_name,200(r1)

  depositor ←depositor ∪∏customer_name, loan_number (r1)

Updating

- A mechanism to change a value in a tuple without changing all values in the tuple
- Use the generalized projection operator to do this task

  $r \leftarrow \prod_{f1,f2 \ldots fr}(r)$

Update examples

- Make interest payments by increasing all balances by 5 percent

  account ←∏account_number, branch_name, balance * 1.05(account)

- Pay all accounts with balances over $10,000 6 percent interest and pay all others 5 percent

  account←∏account_number, branch_name, balance * 1.06(σBAL >10000 (account))

  ∪∏account_number, branch_name, balance * 1.05 (σBAL ≤10000 (account))

12. Discuss briefly about aggregate functions with example.

- **Aggregation function** takes a collection of values and returns single values a result.

  Avg : average value

  Min : minimum value

  Max : maximum value

  Sum : sum of values

  Count : number of values

- Aggregate operation in relational algebra

$G_1,G_2 \ldots G_N \theta F_1(A_1),F_2(A_2) \ldots F_N(A_N)(E)$

E is any relational algebra expression

G1,G2...,Gn is a list of at tributes on which to group(can be empty)

Each Fi is an aggregate function

Each Ai is an attribute name

- **Aggregate function Examples**

Relation r

Empname	Empid	salary
Arun	1001	4000
Vinoth	1002	5000
Ganesh	1003	7000
kaviya	1004	3500

$g_{sum}(salary)(r)$

Sum(salary)
19500

Relational account grouped by branch-name

Branch-name	Account_no	balance
A1	A-102	900
A1	A-304	700
B1	A-207	800
B1	A-206	500
C1	A-207	750

branch_name g sum(balance)(account)

Branch-name	Sum(balance)
A1	1600
B1	1300
C1	750

Result of aggregation does not have a name

- Can use rename operation to give it a name
- For convenience, we permit renaming as part of aggregate operation

branch_name gsum(balance) as sum_balance(account)

13. Discuss in detail about fundamental relational algebra operations

Fundamental Operations

- The select, project, and rename operations are called *unary* operations, because they operate on one relation.

- The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

The Select Operation

- The **select**operation selects tuples that satisfy a given predicate.
- We use the lowercase Greek letter sigma (σ) to denote selection.

Relation r

A	B	C	D
α	α	1	9
α	β	5	7
β	β	67	12
β	β	8	9

$\sigma_{A=B \wedge D>5}(r)$

A	B	C	D
α	α	1	9
β	β	67	12

Notation:$\sigma_p(r)$

- P is called the selection predicate
- Defined as:

  $\sigma_p(r)=\{t|t \in r$ and $p(t)\}$
- Where p is a formula in propositional calculus consisting of terms connected by:∧(and),∨(or),¬(not) Each term is one of:

  <attribute>op<attribute>or<constant>

  Where op is one of:=,≠,>,≥.<.≤n

Example of selection:

σbranch_name="Perryridge"(account)

Project operation

- The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated.
- Projection is denoted by the uppercase Greek letter pi (∏).

Project Example:

Relation r

A	B	C
α	10	1
α	20	1

∏A,C(r)

A	C
α	1

β	30	1
β	40	2

β	1
β	2

Notation:$\prod A_1,A_2 \ldots A_n(r)$

- Where A1,A2 are attribute names and r is a relation name
- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are set

- Example: To eliminate the branch_name attribute of account

  $\prod$**account_number,balance(account)**

Set Union operation

- The union operation is a binary operation, It is used to combined tuples of the given two relations

  Relation r and s:

r

A	B
xxx	12
Yyy	15
Hhh	90

s

A	B
Kkk	12
Jjj	89
Hhh	90

r∪s

A	B
Xxx	12
Yyy	15
Hhh	90
Kkk	12
Jjj	90

- Notation:r∪s
- Defined as: **r ∪ s= {t | t ∈ r or t ∈ s}**

For r∪s to be valid,

1. r,s must have the same arity(same number of attributes)
2. The attribute domains must be compatible (example:2nd column of r deals with the same Type of values as does the 2nd column of s)

Example: to find all customers with either an account or a loan

$\prod_{\text{customer\_name}}$**(depositor)**∪$\prod_{\text{customer\_name}}$**(borrower)**

Set difference operation

It is a binary operation. It is used to find the tuples belongs to one relation not belongs to another relation

r **s** **r-s**

A	B
Xxx	12
Yyy	15
Hhh	90

A	B
Kkk	12
Jjj	89
Hhh	90

A	b
Xxx	12
Yyy	15

- Notation:r - s
- Defined as: **r - s= {t | t ∈ r or t ∉ s}**
- Set difference must be taken between compatible relations
    - r,s must have the **same** arity
    - The attribute domains of r and s must be compatible

Example: To find all customers who have an account not a loan

$\prod_{customer\_name}$**(depositor)**$\cup\prod_{customer\_name}$**(borrower)**

Cartesian product

It is binary operation in fundamental relational algebra it's used to combine the information of the given two relations

Relation r Relation r R X S

A	B
α	1
β	2

C	D	E
α	10	a
β	20	b
γ	30	b

A	B	C	D	E
α	1	α	10	a
α	1	β	20	b
α	1	γ	30	b
β	2	α	10	a
β	2	β	20	b
β	2	γ	30	b

Rename operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$\rho_x(E)$

returns the expression E under the name X

- If a relational-algebra expression *E* has arity *n*, then

$\rho_{x(A_1, A_2, ..., A_n)}(E)$

returns the result of expression *E* under the name *X*, and with the attributes renamed to A_1 , A_2 , …., A_n .

14. Discuss briefly about the additional algebra operations and extended relational algebra operations

Additional relational algebra operation:

- Set intersection
- Natural join
- Outer Join
- Division

➢ All above, other than aggregation**,** can be expressed using basic operations we have seen earlier

Set intersection(∩)

- The union operation is a binary operation, It is used to extract common tuples of the given two relations

Relation r

A	B
xxx	12
Yyy	15
Hhh	90

Relation s

A	B
Kkk	12
Jjj	89
Hhh	90

r∩s

A	b
Hhh	90

- Notation: r∪s
- Defined as: **r ∪ s= {t | t ∈ r and t ∈ s}**

For r∪s to be valid.

3. r,s must have the same arity(same number of attributes)

Example: to find all customers with either an account or a loan

$\prod_{customer\_name}$(depositor)∩$\prod_{customer\_name}$(borrower)

Natural join

➢ **Notation: r ⋈ s**

Relations r

A	C
α	10
β	30
γ	30

Relations s

A	C	D	B
α	10	a	13
β	20	b	16
γ	30	b	16

➢ R ⋈ S:

A	B	C	D
α	13	10	a
γ	16	30	b

- Let *r* and *s* be relations on schemas *R* and *S* respectively. Then, r s is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from *r* and t_s from *s*.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple *t* to the result, where
    - *t* has the same value as t_r on *r*
    - *t* has the same value as t_s on *s*
  - Example:

$R = (A, B, C, D)$ $S = (E, B, D)$

- Result schema = (A, B, C, D, E)
- $r \bowtie s$ is defined as:

$$\prod_{r.A,\, r.B,\, r.C,\, r.D,\, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

Division operation

- Notation: r ÷s
- Suited to queries that include the phrase "for all".
- Let r and s be relations on schemas R and S respectively where
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$
- The result of r ÷ s is a relation on schema

$R - S = (A_1, \ldots, A_m)$

$r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$

Where tu means the concatenation of tuples t and u to produce a single tuple

r

A	C	D	E
Xxx	12	a	1
yyy	15	b	1
Hhh	90	b	3
Xxx	12	b	1
Hhh	90	a	3

s

D	E
a	1
b	1

r ÷s

A	b
Xxx	12

Extended Relational Algebra operations

- Outer join
- Aggregate functions
- Generalized projection

Outer join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.
- Relation *loan*

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

- Join

  loan $\bowtie$ borrower

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- Left Outer Join

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

- Right outer join

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Null values:

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)

Unit III

2 Marks

1. **What is SQL data definition?**

   The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

   - The schema for each relation.
   - The types of values associated with each attribute.
   - The integrity constraints.
   - The set of indices to be maintained for each relation.
   - The security and authorization information for each relation.
   - The physical storage structure of each relation on disk.

2. **Define null values.**

   - It is possible for tuples to have a null value, denoted by *null*, for some of their attributes *null* signifies an unknown value or that a value does not exist.
   - The predicate is null can be used to check for null values.
   - Example: Find all loan number which appears in the *loan* relation with null values for *amount.*
     - **Select** *loan_number***from** l*oan***where** *amount* **is null**

3. **List the Basic domain type in SQL.**

   - char(n). Fixed length character string, with user-specified length *n.*
   - varchar(n). Variable length character strings, with user-specified maximum length *n.*
   - int. Integer (a finite subset of the integers that is machine-dependent).
   - smallint. Small integer (a machine-dependent subset of the integer domain type).
   - numeric (p,d). Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
   - real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.
   - float(n). Floating point number, with user-specified precision of at least *n* digits.

4. **What is meant by set operation in SQL?**

   - The set operations union, intersect, and except operate on relations and correspond to the relational algebra operations $\cup, \cap, -$.

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions union all, intersect all and except all. Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then, it occurs:
  - *m* + *n* times in *r* union all *s*
  - min(*m*,*n)* times in *r*intersect all *s*
  - max(0, *m* – *n)* times in *r*except all *s*

5. What is a tuple variable in SQL?

- Tuple variables are defined in the **from** clause via the use of the as clause.
- Find the customer names and their loan numbers and amount for all customers having a loan at some branch.
- Select *customer_name,T.loan_number,S.amount* from *borrower* as *T, loan* as *S* where *T.loan_number = S.loan_number*
- Find the names of all branches that have greater assets than some branch located in brooklyn.

  Select distinct *T.branch_name*

  from *branch* **as** *T, branch* **as** *S*

  where *T.assets* > *S.assets* **and** *S.branch_city* = 'Brooklyn'
- Keyword**s** is optional and may be omitted *borrower* **as** *T* ≡ *borrowerT*
- Some database such as Oracle *require***as** to be omitted

6. What is a schema definition?

- Define an SQL relation by using the **create table** command.
- Syntax to create the relation

  create table r
  (A1 D1,
  A2 D2,
  . . . ,
  An Dn,
  integrity-constraint$_1$,
  . . . ,
  integrity-constraint$_k$);
- The followingcommand creates a relation *department* in the database.

  create table *department*

(*dept name* **varchar** (20),

building **varchar** (15),

budget **numeric** (12,2),

primary key (*dept name*));

7. **How to remove the relation from SQL database?**
   - To remove a relation from an SQL database, we use the **drop table** command.
   - The **drop table** command deletes all information about the dropped relation fromthe database.
   - The command is

     drop table *r*;

   - Delete is also a more drastic action than

     delete from *r*;

8. **How to change the relation in SQL database?**
   - The **alter table** command to add attributes to an existing relation. Alltuples in the relation are assigned *null* as the value for the new attribute.
   - The formof the **alter table** command is

     alter table *r* **add** *AD*;

     where *r* is the name of an existing relation, *A* is the name of the attribute to beadded, and *D* is the type of the added attribute.

   - We can drop attributes from arelation by the command

     alter table *r* **drop** *A*;

     where *r* is the name of an existing relation, and *A* is the name of an attribute of therelation.

   - Many database systems do not support dropping of attributes, althoughthey will allow an entire table to be dropped.

9. **What is the Select clause?**
   - The **select** clause list the attributes desired in the result of a query.
   - Corresponds to the projection operation of the relational algebra
   - Example: find the names of all branches in the *loan* relation:

     select *branch_name* **from** *loan*

   - SQL allows duplicates in relations as well as in query results.

- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- The keyword **all** specifies that duplicates not be removed.

  select all*branch_name* **from** *loan*

- An asterisk in the select clause denotes "all attributes"

  select ***from** *loan*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
- E.g.: **select***loan_number, branch_name, amount* * 100 **from** *loan*

10. Explain The Where clause

- The where clause specifies conditions that the result must satisfy
- Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

  select *loan_number* from *loan*
  where *branch_name* ='Perryridge'and *amount* > 1200
- Comparison results can be combined using the logical connectives and, or, and not.

11. Explain The from clause

- The **from** clause lists the relations involved in the query
- Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower X loan*

  select ***from** *borrower, loan*
- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

select *customer_name, borrower.loan_number, amount*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number* **and***branch_name* = 'Perryridge

12. Explain The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

  select *customer_name, borrower.loan_number* **as** *loan_id, amount*
  from *borrower, loan*
  where ***borrower.loan_number = loan.loan_number***

5 Marks

13. Discuss briefly about several parts of SQL language.

The SQL language has several parts:

- **Data-definition language** (DDL). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language** (DML). The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity**. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition**. The SQL DDL includes commands for defining views.
- **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

14. Write about the basic schema definition in SQL data definition

- The set of relations in a database must be specified to the system by means of adata-definition language (DDL). The SQL DDL allows specification of not only aset of relations, but also information about each relation
- We can define an SQL relation by using the **create table** command.

  Syntax to create the relation

  create table r
  (A1 D1,
  A2 D2,
  . . . ,

An Dn,

integrity-constraint$_1$,

. . . ,

integrity-constraint$_k$);

- The following command creates a relation *department* in the database.

create table *department*

(*dept name* **varchar** (20),

building **varchar** (15),

budget **numeric** (12,2),

primary key (*dept name*));

- **Remove the relation**
    1. To remove a relation from an SQL database, we use the **drop table** command.
    2. The **drop table** command deletes all information about the dropped relation from the database. The command is

        drop table *r*;
    3. Delete is also a more drastic action than

        delete from *r*;
- **Change the relation**
    1. The **alter table** command to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute.
    2. The form of the **alter table** command is

        alter table *r* **add** *AD*;

        where *r* is the name of an existing relation, *A* is the name of the attribute to be added, and *D* is the type of the added attribute.
    3. We can drop attributes from a relation by the command

        alter table *r* **drop** *A*;

        where *r* is the name of an existing relation, and *A* is the name of an attribute of the relation.
    4. Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

10 Marks

15. Explain Select statement with all possible options

- The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.1 The *instructor* relation.

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.

- **Queries on a Single Relation**

  select *name* **from** *instructor*;

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

"**select** *name* **from** *instructor*".

- "Find the names of all instructors in the Computer Science department who have salary greater than $70,000."

```
select name
from instructor
where dept name = 'Comp. Sci.'
and salary > 70000
```

name
Katz
Brandt

- "Retrieve the names of all instructors, along with their department names and department building name."

select name, instructor.dept name, building
from instructor, department
where instructor.dept name= department.dept name;

name	dept_name	building
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson

- Cartesian Product

select name, course id
from instructor, teaches
where instructor.ID= teaches.ID;

inst.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

- "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

select name, course id
from instructor, teaches
where instructor.ID= teaches.ID and instructor.dept name = 'Comp. Sci.';

name	course_id
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

- Natural join similar to Cartesian product

select name, course id
from instructor natural join teaches;
Or
select name, title
from (instructor natural join teaches) join course using (course id);

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010

16. Discuss briefly about the Basic structure of SQL and rename operation.

- The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**.

- The query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result.
- We introduce the SQL syntax through examples,
- A typical SQL query has the form:

  select A_1, A_2, ..., A_n

  from r_1, r_2, ..., r_m

  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- This query is equivalent to the relational algebra expression.
- The result of an SQL query is a relation.
- **<u>The Select clause</u>**
  - The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
  - Example: find the names of all branches in the *loan* relation:

    select *branch_name* **from** *loan*
  - In the relational algebra, the query would be:

$\pi_{branch\_name}(loan)$

  - NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

    E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*

    Some people use upper case wherever we use bold font.
  - SQL allows duplicates in relations as well as in query results.
  - To force the elimination of duplicates, insert the keyword **distinct** after select**.**
  - Find the names of all branches in the *loan* relations, and remove duplicates

    select distinct *branch_name* **from** *loan*
  - The keyword **all** specifies that duplicates not be removed.

    select all*branch_name* **from** *loan*
  - An asterisk in the select clause denotes "all attributes"

    select ***from** *loan*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
- E.g.: **select***loan_number, branch_name, amount* * 100 **from** *loan*

The Where clause

- The where clause specifies conditions that the result must satisfy. Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

  select *loan_number* from *loan*
  where *branch_name* ='Perryridge'and *amount* > 1200
- Comparison results can be combined using the logical connectives and, or, and not.

The from clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower X loan*

  select ***from** *borrower, loan*
- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

  select *customer_name, borrower.loan_number, amount* **from** *borrower, loan* **where** *borrower.loan_number = loan.loan_number* **and** *branch_name* = 'Perryridge

The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:
  - *old-name* **as** *new-name*
- E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*
  - **select** *customer_name, borrower.loan_number* **as** *loan_id, amount* **from** *borrower, loan* **where *borrower.loan_number = loan.loan_number***

17. Discuss briefly about the string operation in SQL.

- SQL specifies strings by enclosing them in single quotes, for example, 'Computer'.
- A single quote character that is part of a string can be specified by using two single quote characters; for example, the string "It's right" can be specified by "It"s right".
- The SQL standard specifies that the equality operation on strings is case sensitive; as a result the expression "'comp. sci.' = 'Comp. Sci.'" evaluates to false.
- SQL also permits a variety of functions on character strings, such as concatenating (using "_"), extracting substrings, finding the length of strings, converting strings to uppercase (using the function **upper**(*s*) where *s* is a string) and lowercase (using the function **lower**(*s*)), removing spaces at the end of the string(using **trim**(*s*)) and so on.
- There are variations on the exact set of string functions supported by different database systems. See your database system's manual for more details on exactly what string functions it supports. Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:
    - Percent (%): The % character matches any substring.
    - Underscore ( ): The character matches any character.
- Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:
    - 'Intro%' matches any string beginning with "Intro".
    - '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
    - ' ' matches any string of exactly three characters.
    - ' %' matches any string of at least three characters.
- SQL expresses patterns by using the **like** comparison operator.
- Consider the query "Find the names of all departments whose building name includes the substring'Watson'." This query can be written as:

```
select dept name
from department
where building like '%Watson%';
```

18. Explain Set operation, Aggregate operation in SQL.

SET operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −. We shall now construct queries involving the **union**, **intersect**, and **except** operations over two sets.

The Union Operation

- Find all customers who have a loan, an account, or both:

  (select*customer_name* **from** *depositor*)
  union
  (select*customer_name* **from** *borrower)*

- The **union** operation automatically eliminates duplicates, unlike the **select** clause.
- If we want to retain all duplicates, we must write **union all** in place of **union**:

  (select*customer_name* **from** *depositor*)
  union all
  (select*customer_name* **from** *borrower)*

The intersection operation

- Find all customers who have both a loan and an account.

  (**select***customer_name* **from** *depositor*)
  intersect
  (select*customer_name* **from** *borrower)*

- The **intersect**operation automatically eliminates duplicates, unlike the select clause.
- If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

  (**select***customer_name* **from** *depositor*)
  intersect all
  (select*customer_name* **from** *borrower)*

Set difference or Except operation

- Find all customers who have an account but no loan.

  (**select***customer_name* **from** *depositor*)
  except
  (select*customer_name* **from** *borrower)*

- The **except**operation automatically eliminates duplicates, unlike the select clause.
- If we want to retain all duplicates, we must write **except all** in place of **except**:

(**select***customer_name* **from** *depositor*)
except all
(select*customer_name* **from** *borrower)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg:average value

  min: minimum value

  max: maximum value

  sum: sum of values

  count:number of values

- Find the average account balance at the Perryridge branch.

  select avg *(balance)* **from** *account*
  where *branch_name* = 'Perryridge'

- Find the number of tuples in the *customer* relation.

  select count (*) **from** *customer*

- Find the number of depositors in the bank.

  select count (distinct *customer_name)* **from** *depositor*

Aggregation with grouping

- The aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause.
- The attribute or attributes given in the **group by** clause are used to form groups.
- Tuples with the same value on all attributes in the **group by** clause are placed in one group.

Example

- Find the number of depositors for each branch.

  select *branch_name,* **count (distinct***customer_name)*
  from *depositor, account*
  where *depositor.account_number = account.account_number*
  group by *branch_name*

The Having Clause

- It is useful to state a condition that applies to groups rather than to tuples.

- we use the **having** clause of SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used.
- We express this query in SQL as follows:

  Find the names of all branches where the average account balance is more than $1,200.

```
    select branch_name, avg (balance)
from account
group by branch_name
having avg(balance)>1200
```

19. Explain in detail about the nested sub query and complex query in sql.

Nested Sub queries

- SQL provides a mechanism for nesting sub queries.
- A sub query is a **select-from where** expression that is nested within another query.
- A common use of sub queries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting sub queries in the **where** clause.

Set Membership

- SQL allows testing tuples for membership in a relation.
- The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause.
- The **not in** connective tests for the absence of set membership.

  Example for **in** and **not in** construct

  Find all customers who have both an account and a loan at the bank

```
select distinct customer_name
from borrower
where customer_name in (select customer_namefromdepositor )
```

Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
from borrower
where customer_name not in (select customer_name from depositor )
```

Set Comparison

As an example of the ability of a nested sub query to compare sets.

Find all branches that have greater assets than some branch located in Thanjavur.

select distinct *T.branch_name*
from *branch* **as** *T, branch* **as** *S*
where *T.assets > S.assets* **and***S.branch_city = 'thanjavur*'

Same query using >**some** clause

select *branch_name* **from** *branch*
where *assets* >**some** **(select** *assets* **from** *branch* **where** *branch_city = thanjavur*')

"All" Construct

Find the names of all branches that have greater assets than all branches located in thanjavur

select *branch_name***from** *branch***where** *assets* >**all**
(select *assets***from** *branch***where** *branch_city* = 'thanjavur')

Complex queries – Sub-queries in the From Clause

- SQL allows a sub-query expression to be used in the **from** clause.
- The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.
- Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000." We wrote this query in using the **having** clause.
- We can now rewrite this query, without using the **having** clause, by using a sub query in the **from** clause, as follows:

  select *dept name*, *avg salary*
  from (**select** *dept name*, **avg** (*salary*) **as** *avg_ salary*
  from *instructor*
  group by *dept name*)
  where *avg salary* >42000;

- The sub query generates a relation consisting of the names of all departments and their corresponding average instructors' salaries.
- The attributes of the sub query result can be used in the outer query, as can be seen in the above example.
- As another example, suppose we wish to find the maximum across all departments of the total salary at each department. The **having** clause does not help us in this task, but we can write this query easily by using a sub query in the **from** clause, as follows:

  select max (*tot salary*)
  from (**select** *dept name*, **sum**(*salary*)

from *instructor*
group by *dept name*) **as** *dept total* (*dept name, tot salary*);

The with Clause

- The **with** clause provides away of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Consider the following query, which finds those departments with the maximum budget.

with *max budget* (*value*) **as**
(**select max**(*budget*)
from *department*)
select *budget*
from *department, max budget*
where *department.budget = max budget.value*;

- The **with** clause defines the temporary relation *max budget,* which is used in the immediately following query.
- The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.
- We could have written the above query by using a nested sub query in either the **from** clause or the **where** clause.
- However, using nested sub queries would have made the query harder to read and understand.
- The **with** clause makes the query logic clearer; it also permits a view definition to be used in multiple places within a query.
- For example, suppose we want to find all departments where the total salary is greater than the average of the total salary at all departments. We can write the query using the **with** clause as follows.

with *dept total* (*dept name, value*) **as**
(**select** *dept name*, **sum**(*salary*)
from *instructor*
group by *dept name*),
dept total avg(*value*) **as**
(**select avg**(*value*)
from *dept total*)
select *dept name*
from *dept total, dept total avg*
where *dept total.value >= dept total avg.value*;

- We can, of course, create an equivalent query without the **with** clause, but it would be more complicated and harder to understand.

20. Discuss briefly about Authorization.

Authorization

- We may assign a user several forms of authorizations on parts of the database.

Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

- Each of these types of authorizations is called a **privilege**.
- We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view

Granting and Revoking of Privileges

- The SQL standard includes the privileges **select**, **insert**, **update**, and **delete**.
- The privilege **all privileges** can be used as a short form for all the allowable privileges.
- user who creates a new relation is given all privileges on that relation automatically.
- The SQL data-definition language includes commands to grant and revoke privileges.
- The **grant** statement is used to confer authorization.
- The basic form of this statement is:

  grant <privilege list>
  on <relation name or view name>
  to <user/role list>;

- The *privilege list* allows the granting of several privileges in one command.
- The **select** authorization on a relation is required to read tuples in the relation.
- The following **grant** statement grants database users Amit and Satoshi

  Select authorization on the *department* relation:

  - **grant select on** *department* **to** *Amit*, *Satoshi*;

- To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

  revoke <privilege list>
  on <relation name or view name>
  from <user/role list>;

- Thus, to revoke the privileges that we granted previously, we write

  revoke select on *department* **from** *Amit, Satoshi*;

  revoke update (budget) **on** *department* **from** *Amit, Satoshi*;

- Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user.

21. Discuss about Integrity constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.

Thus, integrity constraints guard against accidental damage to the database.

Examples of integrity constraints are:

- An instructor name cannot be *null*.
- No two instructors can have the same instructor ID.
- Every department name in the *course* relation must have a matching department name in the *department* relation.
- The budget of a department must be greater than $0.00.

Constraints on a Single Relation

- **Unique**

  unique (*Aj*1 , *Aj*2, . . . , *Ajm*)

- **not null**

  name **varchar**(20) **not null**

  budget **numeric**(12,2) **not null**

Check constraint

- A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system.
- For instance, a clause **check** (*budget* > 0) in the **create table** command for relation department *would* ensure that the value of **budget** is nonnegative.

As another example, consider the following:

create table *section*
(*course id* **varchar** (8),
building **varchar** (15),
room number **varchar** (7),
time slot id **varchar** (4),

primary key (*course id, sec id, semester, year*),
check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer')));

- **primary key (A_1, ..., A_n)**

Example: Declare *branch_name* as the primary key for *branch*

```
create table branch
(branch_name char(15),
branch_city char(30) not null,
assets integer,
primary key (branch_name))
```

22. Discuss in detail about the modification of database and joining a relation.

Modification of the database

- In the modification of database we show how to add, remove, or change information with SQL.

Deletion

- A delete request is expressed in much the same way as a query.
- We can delete only whole tuples; we cannot delete values on only particular attributes.
- SQL expresses a deletion by

  delete from *r* **where** *P*;

- **Where***P* represents a predicate and *r* represents a relation.
- The **delete** statement first finds all tuples *t* in *r* for which *p* (*t*) is true, and then deletes them from *r*. The **where** clause can be omitted, in which case all tuples in *r* are deleted.
- A **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.
- The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the **where** clause may be empty. The request

  delete from *instructor*;

deletes all tuples from the *instructor* relation. The *instructor* relation itself still exists, but it is empty.

- Here are examples of SQL delete requests:

  Delete all account tuples at the Perryridge branch

  delete from *account*

  where *branch_name* = 'Perryridge'

- Delete all accounts at every branch located in the city 'Needham'.

delete from *account*
where *branch_name* **in** (**select** *branch_name***from** *branch*
where *branch_city* = 'Needham')

- Delete the record of all accounts with balances below the average at the bank.

delete from *account*
where *balance* < (**select avg** (*balance*)**from** *account*)

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:

1. First, compute **avg** balance and find all tuples to delete
2. Next, delete all tuples found above (without re-computing **avg** or retesting the tuples)

Insertion

- To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.
- The attribute values for inserted tuples must be members of the corresponding attribute's domain.
- Similarly, tuples inserted must have the correct number of attributes.
- The simplest **insert** statement is a request to insert one tuple.
- Add a new tuple to *account*

insert into *account***values** ('A-9732', 'Perryridge', 1200)

or equivalently

insert into *account* (*branch_name, balance, account_number*)
values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

insert into *account*
values ('A-777','Perryridge', *null*)

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account
  - **insert into** *account*
    select *loan_number, branch_name,* 200
    from *loan*
    where *branch_name* = 'Perryridge'

- **insert into** *depositor*
  select *customer_name, loan_number*
  from *loan, borrower*
  where branch_name = 'Perryridge'
  and *loan.account_number = borrower.account_number*

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation

  insert into*table*1 **select** * **from***table*1

Updates

- We may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used.
- As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query.
- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

    update *account*
    set *balance* = *balance* * 1.06
    where *balance* > 10000
  - The order is important
  - Can be done better using the **case** statement
  - Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

update*account*
set*balance* = **case**
when*balance*<= 10000 **then***balance* *1.05
else*balance* * 1.06
end

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as sub query expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using (A_1, A_1, …, A_n)

- Relation *loan*&Relation *borrower*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- ***Inner Join***

  loan **inner join** *borrower*

  on *loan.loan_number = borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- **Left Outer Join**

  loan **left outer join** *borrower* **on** *loan.loan_number = borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

- **Natural Join**
  - *loan* **natural inner join***borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- **Right Outer Join**
  - *loan* **natural right outer join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Unit IV

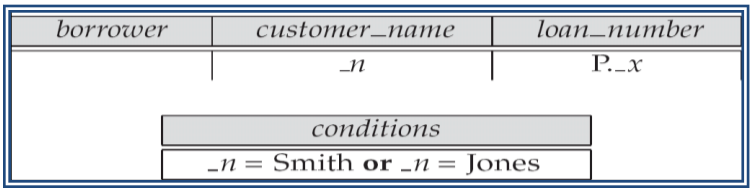
2 Marks

1. **Define tuple relational calculus**.
   - The tuple relational calculus is a nonprocedural query language.
   - It describes the desired information without giving a specific procedure for obtaining that information.
   - A query in the tuple relational calculus is expressed as:**{*t* | *P*(*t*)}**
     That is, it is the set of all tuples *t* such that predicate *P* is true for *t*.
2. **What is meant by a domain relational calculus?**
   - A nonprocedural query language equivalent in power to the tuple relational calculus
   - Each query is an expression of the form:
     $\{ <x_1, x_2, \ldots, x_n> \mid P(x_1, x_2, \ldots, x_n)\}$
     - $x_1, x_2, \ldots, x_n$ represent domain variables
     - *P* represents a formula similar to that of the predicate calculus
3. **What is QBE?**
   - A graphical query language which is based (roughly) on the domain relational calculus
   - **Two dimensional syntax** – system creates templates of relations that are requested by users
   - Queries are expressed "by example"
   - Find all loan numbers at the Perryridge branch.

loan	*loan_number*	*branch_name*	*amount*
	P._*x*	Perryridge	

     - x is a variable (optional; can be omitted in above query)
     - P. means print (display)
     - duplicates are removed by default
     - To retain duplicates use P.ALL
4. **Define Entity Sets.**
   - An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity
   - An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.
   - An **entity set** is a set of entities of the same type that share the same properties, or attributes.
5. **Define the purpose of Condition box in QBE.**
   - Allows the expression of constraints on domain variables that are either inconvenient or impossible to express within the skeleton tables.

- Complex conditions can be used in condition boxes
- Example: Find the loan numbers of all loans made to Smith, to Jones, or to both jointly



6. **What are the design phases in SQL?**
   - The design phases are
     i. Initial phase
     ii. Conceptual design phase
     iii. Specification of functional requirements
     iv. Final design phase
7. **What is an E-R model?**
   - The entity relationship model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
8. **What are the basic notions in E-R model?**
   - The basic notions are
     i. Entity sets
     ii. Relationships sets
     iii. Attributes

<u>5 Marks / 10 Marks</u>

9. **Explain the design phases in SQL?**
   - Design process may be easy for small applications. But for real world applications they are often complex.
   - The database designer must interact with users of the application to understand the needs of the application represent them in high-level fashion that can be understood by the user and then translate the requirement into lower levels of the design
   - The design phases are
     i. Initial phase
     ii. Conceptual design phase
     iii. Specification of functional requirements
     iv. Final design phase
   - Initial phase

- The initial phase of database design is to characterize fully the data need of the database users.
- The database designer need to interact with domain experts and users to came out this task.
- The outcome of this phase is a specification of user requirement.

➢ Conceptual design phase
- The designer chooses a data model and by applying the concepts of the chosen data model, translated the requirement into a conceptual schema of the database
- The schema provides a details overview of the enterprise

➢ Specification of functional requirements
- In this stage the designer review the schema to ensure it meets functional requirements which describes operation such as
  1. Modifying or updating data
  2. Searching for and retrieving specific data
  3. Deleting data

➢ Final design phase
- The process of moving from an abstract data model to the implementation of the database proceeds in two final designs.
  1. Logical design phase
     - The designer maps the high level conceptual schema into the implementation data model of the database system that will be used.
     - This step typically maps the conceptual schema defined using the ER model into a relation schema
  2. Physical design phase
     - The designer uses the resulting system specific database schema in the subsequent physical design phase, in which the physical features such as form of file organization and the internal storage structures are specified

10. Explain briefly about Tuple relational calculus with example.

- A nonprocedural query language, where each query is of the form
  - $\{t \mid P(t)\}$
- It is the set of all tuples *t* such that predicate *P* is true for *t*

Predicate Calculus Formula

➢ Set of attributes and constants

- Set of comparison operators: (e.g., $<, \leq, =, \neq, >, \geq$)
- Set of connectives: and ($\wedge$), or (v), not ($\neg$)
- Implication ($\Rightarrow$): x $\Rightarrow$ y, if x if true, then y is true
  $x \Rightarrow y \equiv \neg x$ v y
- Set of quantifiers:
  - $\exists\, t \in r\ (Q\ (t\ )) \equiv$"there exists" a tuple in t in relation r such that predicate $Q\ (t\ )$ is true
  - $\forall t \in r\ (Q\ (t\ )) \equiv Q$ is true "for all" tuples t in relation r

Example Queries

- Find the *loan_number, branch_name,* and *amount* for loans of over $1200
  $\{t \mid t \in loan \wedge t\ [amount\ ] > 1200\}$
- Find the loan number for each loan of an amount greater than $1200
  $\{t \mid \exists\ s\ \in$loan $(t\ [loan\_number\ ] = s\ [loan\_number\ ] \wedge s\ [amount\ ] > 1200)\}$
- Find the names of all customers having a loan, an account, or both at the bank
  $\{t \mid \exists s \in borrower\ (\ t\ [customer\_name\ ] = s\ [customer\_name\ ])$
  $\vee \exists u \in depositor\ (\ t\ [customer\_name\ ] = u\ [customer\_name\ ])$
- Find the names of all customers having a loan at the Perryridge branch
  $\{t \mid \exists s \in borrower\ (t\ [customer\_name\ ] = s\ [customer\_name\ ]$
  $\wedge \exists u \in loan\ (u\ [branch\_name\ ] =$ "Perryridge"
  $\wedge u\ [loan\_number\ ] = s\ [loan\_number\ ]))\}$
- Find the names of all customers having a loan from the Perryridge branch, and the cities in which they live

  $\{t \mid \exists s \in loan\ (s\ [branch\_name\ ] =$ "Perryridge"
  $\wedge \exists u \in borrower\ (u\ [loan\_number\ ] = s\ [loan\_number\ ]$
  $\wedge t\ [customer\_name\ ] = u\ [customer\_name\ ])$
  $\wedge \exists\ v \in customer\ (u\ [customer\_name\ ] = v\ [customer\_name\ ]$
  $\wedge t\ [customer\_city\ ] = v\ [customer\_city\ ]))\}$

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, $\{\ t \mid \neg t \in r\ \}$ results in an infinite relation if the domain of any attribute of relation r is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.

- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P

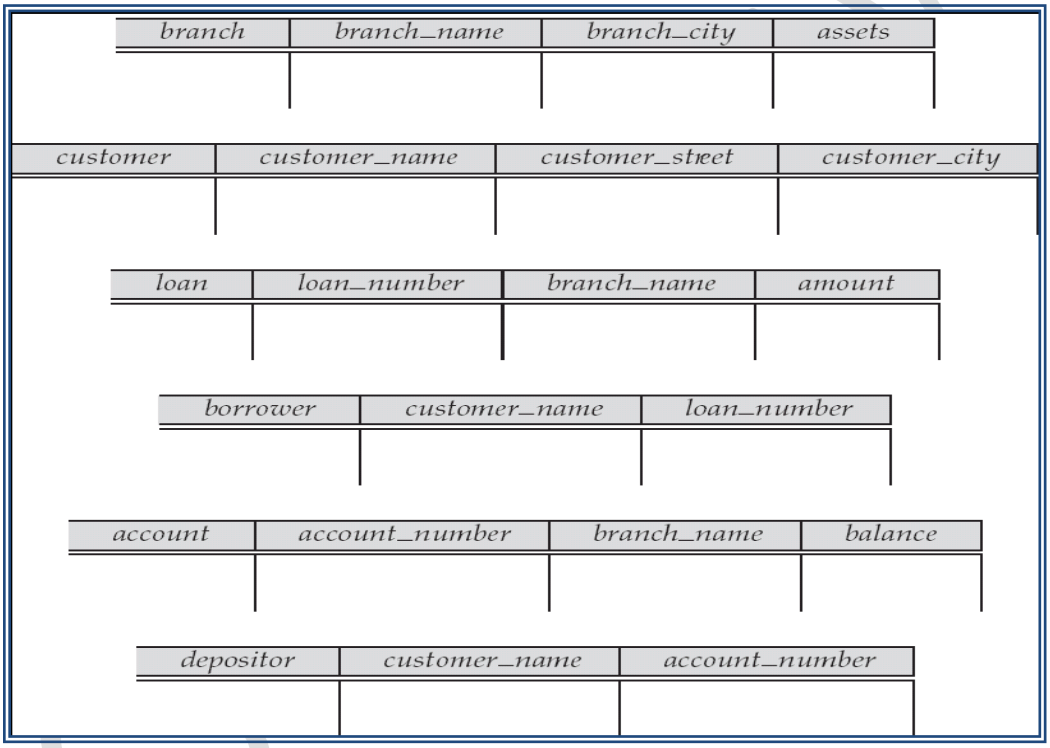
➢ NOTE: this is more than just a syntax condition.

➢ E.g. $\{ t \mid t[A] = 5 \lor \text{true} \}$ is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P.

11. Discuss briefly about QBE with examples.

- A graphical query language which is based (roughly) on the domain relational calculus
- **Two dimensional syntax** – system creates templates of relations that are requested by users
- Queries are expressed "by example"

QBE Skeleton Tables for the Bank Example



Queries on One Relation

- Find all loan numbers at the Perryridge branch.

loan	loan_number	branch_name	amount
	P._x	Perryridge	

➢ _x is a variable (optional; can be omitted in above query)

➢ P. means print (display)

➢ duplicates are removed by default

➢ To retain duplicates use P.ALL

loan	loan_number	branch_name	amount
	P.ALL.	Perryridge	

- Find the loan number of all loans with a loan amount of more than $700

loan	loan_number	branch_name	amount
	P.		>700

- Find names of all branches that are not located in Brooklyn

branch	branch_name	branch_city	assets
	P.	¬ Brooklyn	

- Find the loan numbers of all loans made jointly to Smith and Jones

borrower	customer_name	loan_number
	Smith Jones	P._x _x

- Find all customers who live in the same city as Jones

customer	customer_name	customer_street	customer_city
	P._x Jones		_y _y

Queries on Several Relations

Find the names of all customers who have a loan from the Perryridge branch.

loan	loan_number	branch_name	amount
	_x	Perryridge	

borrower	customer_name	loan_number
	P._y	_x

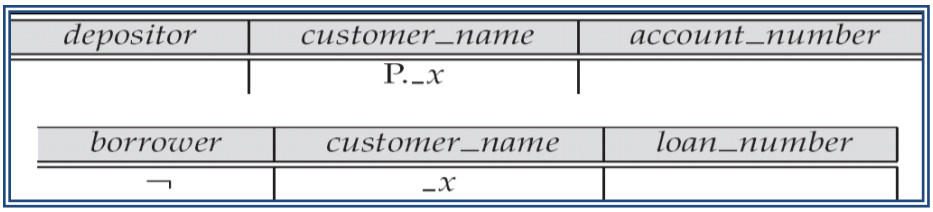
- Find the names of all customers who have both an account and a loan at the bank.

depositor	customer_name	account_number
	P._x	

borrower	customer_name	loan_number
	_x	

Negation in QBE

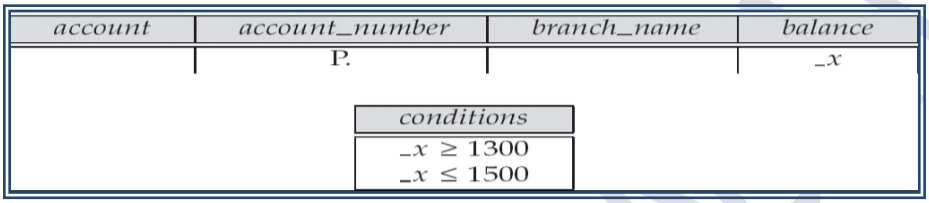
- Find the names of all customers who have an account at the bank, but do not have a loan from the bank.



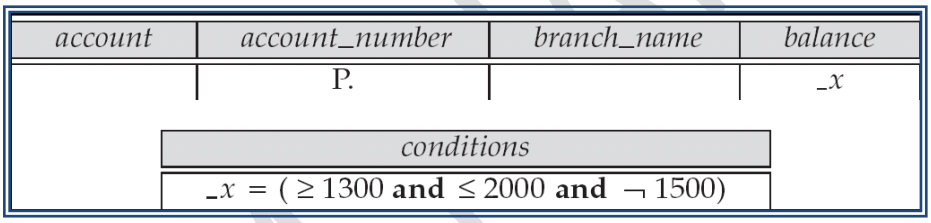
¬ means "there does not exist"

The Condition Box

- Allows the expression of constraints on domain variables that are either inconvenient or impossible to express within the skeleton tables.
- Complex conditions can be used in condition boxes
- Example: Find all account numbers with a balance greater than $1,300 and less than $1,500



- Find all account numbers with a balance greater than $1,300 and less than $2,000 but not exactly $1,500.



The Result Relation Join and projection

- Find the *customer_name*, *account_number,* and *balance* for all customers who have an account at the Perryridge branch.
    - We need to:
        - Join *depositor* and *account.*
        - Project *customer_name, account_number* and *balance.*
    - To accomplish this we:
        - Create a skeleton table, called *result,* with attributes *customer_name, account_number,* and *balance.*
        - Write the query.

The resulting query is:

account	account_number	branch_name	balance
	_y	Perryridge	_z

depositor	customer_name	account_number
	_x	_y

result	customer_name	account_number	balance
P.	_x	_y	_z

Ordering the Display of Tuples

- AO = ascending order; DO = descending order.
- Example: list in ascending alphabetical order all customers who have an account at the bank

depositor	customer_name	account_number
	P.AO.	

- When sorting on multiple attributes, the sorting order is specified by including with each sort operator (AO or DO) an integer surrounded by parentheses.
- Example: List all account numbers at the Perryridge branch in ascending alphabetic order with their respective account balances in descending order.

account	account_number	branch_name	balance
	P.AO(1).	Perryridge	P.DO(2).

12. Discuss briefly about entity-Relationship diagram and mapping cardinality.

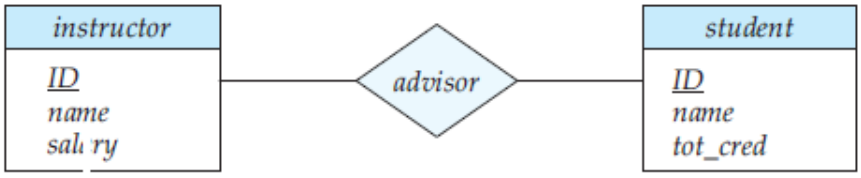
Entity-Relationship Diagrams

E-R diagram can express the overall logical structure of a database graphically.

E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model.

Basic Structure

An E-R diagram consists of the following major components:

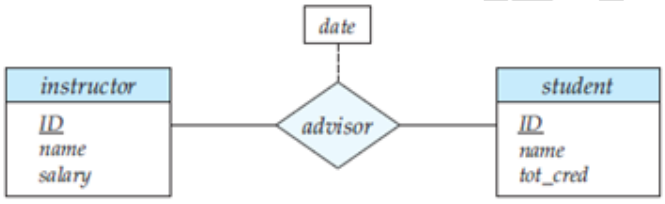


- **Rectangles divided into two parts** represent entity sets.
  - The first part, which in this textbook is shaded blue, contains the name of the entity set.

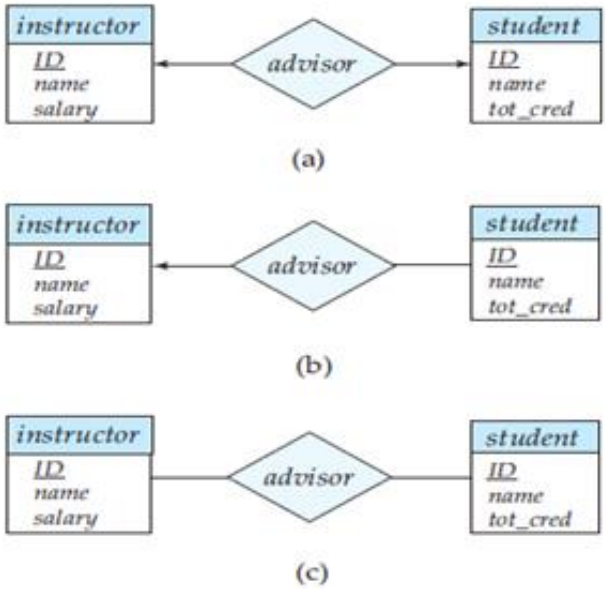
    - The second part contains the names of all the attributes of the entity set.
- **Diamonds** represent relationship sets.
- **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.
- **Lines** link entity sets to relationship sets.
- **Dashed lines** link attributes of a relationship set to the relationship set.
- **Double lines** indicate total participation of an entity in a relationship set.
- **Double diamonds** represent identifying relationship sets linked to weak entity sets

Consider the E-R diagram which consists of two entity sets, *instructor* and *student* related through a binary relationship set *advisor*.

- The attributes associated with *instructor* are *ID*, *name*, and *salary*.
- The attributes associated with *student* are *ID*, *name*, and *tot_cred*.
- Primary keys(ID) are underlined.
- For example, we have the *date* descriptive attribute attached to the relationship set *advisor* to specify the date on which an instructor became the advisor.



- E-R diagram with an attribute attached to a relationship set.



Relationships. (a) One-to-one. (b) One-to-many. (c) Many-to-many.

Mapping Cardinality

The relationship set *advisor*, between the *instructor* and *student* entity sets may be one-to-one, one-to-many, many-to-one, or many-to-many. To distinguish among these types,we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question, as follows:

- **One-to-one:** We draw a directed line from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise at most one student, and a student may have at most one advisor.
- **One-to-many:** We draw a directed line from the relationship set *advisor* to the entity set *instructor* and an undirected line to the entity set *student*. This indicates that an instructor may advise many students, but a student may have at most one advisor.
- **Many-to-one:** We draw an undirected line from the relationship set *advisor* to the entity set *instructor* and a directed line to the entity set *student*. This indicates that an instructor may advise at most one student, but a student may have many advisors.
- **Many-to-many:** We draw an undirected line from the relationship set *advisor* to both entities sets *instructor* and *student*. This indicates that an instructor may advise many students, and a student may have many advisors.

13. Explain in detail about Tuple relational calculus and domain relational calculus with example.

The Tuple Relational Calculus

- When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query.
- The tuple relational calculus, by contrast, is a **nonprocedural** query language.
- It describes the desired information without giving a specific procedure for obtaining that information.
- A query in the tuple relational calculus is expressed as:

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

- That is, it is the set of all tuples t such that predicate P is true for t.

Example Queries

- Find the *ID, name, dept name, salary* for instructors whose salary is greater than$80,000:

  $\{t \mid t \in instructor \wedge t[salary] > 80000\}$

- "Find the instructor *ID* for each instructor with a salary greater than $80,000" as:

  $\{t \mid \exists\ s \in instructor\ (t[ID] = s[ID] \wedge s[salary] > 80000)\}$

In English, we read the preceding expression as "The set of all tuples *t* such that there exists a tuple *s* in relation *instructor* for which the values of *t* and *s* for the *ID* attribute are equal, and the value of *s* for the *salary* attribute is greater than$80,000."

Formal Definition

- A tuple-relational-calculus expression is of the form:

  { *t* | *P*(*t*) } Where *P* is a *formula*.
- Several tuple variables may appear in a formula.
- A tuple variable is said to be a *free variable* unless it is quantified by ∃ or ∀.Thus, in:

  ***t* ∈ *instructor* ∧ ∃ *s* ∈ *department*(*t*[*dept name*] = *s*[*dept name*])**

  t is a free variable.
- Tuple variable *s* is said to be a *bound* variable.
- A tuple-relational-calculus formula is built up out of *atoms*.
- An atom has one of the following forms:
  - $s \in r$, where *s* is a tuple variable and *r* is a relation (we do not allow use of the / ∈operator).
  - $s[x]\ \theta\ u[y]$, where *s* and *u* are tuple variables, *x* is an attribute on which *s* is defined, *y* is an attribute on which *u* is defined, and _ is a comparison operator (<, ≤, =, _=, >, ≥); we require that attributes *x* and *y* have domains whose members can be compared by θ
  - $s[x]\ \theta\ c$, where *s* is a tuple variable, *x* is an attribute on which *s* is defined, $\theta$is a comparison operator, and *c* is a constant in the domain of attribute *x*.
- We build up formulae from atoms by using the following rules:
  - An atom is a formula.
  - If *P*1 is a formula, then so are $\neg P1$ and (*P*1).
  - If *P*1 and *P*2 are formulae, then so are ***P*1 ∨ *P*2, *P*1 ∧ *P*2, and *P*1 ⇒ *P*2.**
  - If *P*1(*s*) is a formula containing a free tuple variable *s*, and *r* is a relation, then

    ∃ *s* ∈ *r* (*P*1(*s*)) and ∀ *s* ∈ *r* (*P*1(*s*))are also formulae.
- In the tuple relational calculus, these equivalences include the following three rules:
  i. $P1 \wedge P2$ is equivalent to $\neg(\neg(P1) \vee \neg(P2))$.
  ii. $\forall t \in r\ (P1(t))$ is equivalent to $\neg \exists t \in r\ (\neg P1(t))$.
  iii. $P1 \Rightarrow P2$ is equivalent to $\neg(P1) \vee P2$.

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, { t | $\neg t \in r$ } results in an infinite relation if the domain of any attribute of relation *r* is infinite

- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P

Domain relational calculus:

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:
  - $\{ \langle x_1, x_2, \ldots, x_n \rangle \mid P(x_1, x_2, \ldots, x_n)\}$
  - $x_1, x_2, \ldots, x_n$ represent domain variables
  - P represents a formula similar to that of the predicatecalculus

Example Queries

- Find the *loan_number, branch_name,* and *amount* for loans of over $1200

  $\{\langle l, b, a \rangle \mid \langle l, b, a \rangle \in loan \wedge a > 1200\}$

- Find the names of all customers who have a loan of over $1200

  $\{\langle c \rangle \mid \exists l, b, a\ (\langle c, l \rangle \in borrower \wedge \langle l, b, a \rangle \in loan \wedge a > 1200)\}$

- Find the names of all customers who have a loan from the b1 branch and the loan amount:

  $\{\langle c, a \rangle \mid \exists l\ (\langle c, l \rangle \in borrower \wedge \exists b\ (\langle l, b, a \rangle \in loan \wedge\ b = \text{“b1”}))\}$

  $\{\langle c, a \rangle \mid \exists l\ (\langle c, l \rangle \in borrower \wedge \langle l, \text{“}b1\text{”}, a \rangle \in loan)\}$

- Find the names of all customers having a loan, an account, or both at the b1 branch:

  $\{\langle c \rangle \mid \exists l\ (\langle c, l \rangle \in borrower$
  $\wedge \exists b,a\ (\langle l, b, a \rangle \in loan \wedge b = \text{“b1”}))$
  $\vee \exists a\ (\langle c, a \rangle \in depositor$
  $\wedge \exists b,n\ (\langle a, b, n \rangle \in account \wedge b = \text{“b1”}))\}$

- Find the names of all customers who have an account at all branches located in Trichy:

  $\{\langle c \rangle \mid \exists\ s,n\ (\langle c, s, n \rangle \in \text{customer}) \wedge$

  $\forall x,y,z\ (\langle x, y, z \rangle \in branch \wedge y = \text{“trichy”}) \Rightarrow$
  $\exists a,b\ (\langle x, y, z \rangle \in account \wedge \langle c,a \rangle \in depositor)\}$

Safety of Expressions

The expression: $\{ \langle x_1, x_2, \ldots, x_n \rangle \mid P(x_1, x_2, \ldots, x_n)\}$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom* (*P*) (that is, the values appear either in *P* or in a tuple of a relation mentioned in *P*).

2. For every "there exists" sub formula of the form $\exists x$ ($P_1(x)$), the sub formula is true if and only if there is a value of *x* in *dom* (P_1) such that $P_1(x)$ is true.

3. For every "for all" sub formula of the form $\forall_x$ (P_1 (x)), the sub formula is true if and only if $P_1(x)$ is true for all values *x* from *dom* (P_1).

14. Discuss in detail about the entity relationship diagram, Entity relationship design issues and database design.

Modeling

- A *database* can be modeled as:
  - a collection of entities,
  - relationship among entities.
- An entity is an object that exists and is distinguishable from other objects.
  - example: specific person, company, event, plant
- Entities have *attributes*
  - Example: people have *names* and *addresses*
- An entity set is a set of entities of the same type that share the same properties.
  - Example: set of all persons, companies, trees, holidays

Entity Sets *customer* and *loan*

Customer ID	Customer name	Customer Street	Customer city
321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

Customer

Loan number	Loan amount
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

Loan

Relationship Sets

- A **relationship** is an association among several entities

- Example:

  Hayes *depositor* A-102

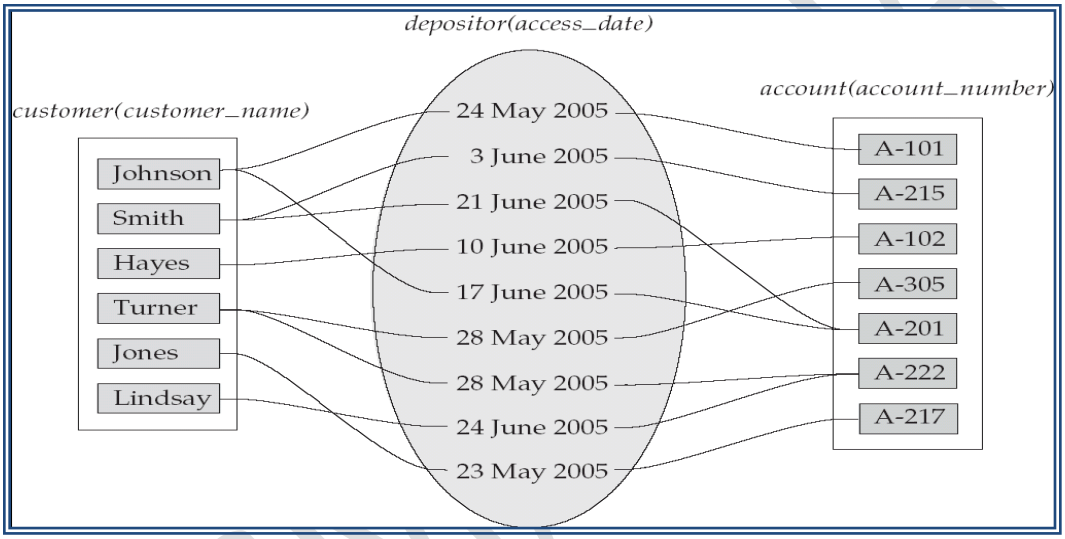
  customer entity relationship set *account* entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

  $\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$

  where $(e_1, e_2, \ldots, e_n)$ is a relationship

  - Example: (Hayes, A-102) $\in$ *depositor*
- An **attribute** can also be property of a relationship set.
- For instance, the *depositor* relationship set between entity sets *customer* and *account* may have the attribute *access-date*

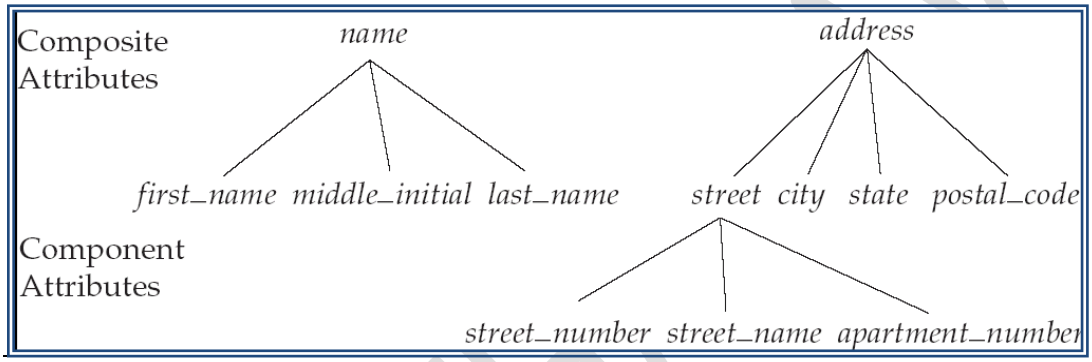


Degree of a Relationship Set

- Refers to number of entity sets that participate in a relationship set.
- Relationship sets that involve two entity sets are **binary** (or degree two). Generally, most relationship sets in a database system are binary.
- Relationship sets may involve more than two entity sets.
  - Example: Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets *employee, job, and branch*
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)

Attributes

- An entity is represented by a set of attributes that is descriptive properties possessed by all members of an entity set.
- **Domain** – the set of permitted values for each attribute
- Attribute types:
  - *Simple* and *composite* attributes.
  - *Single-valued* and *multi-valued* attributes
    - Example: multivalve attribute: *phone_numbers*
  - *Derived* attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth



Keys

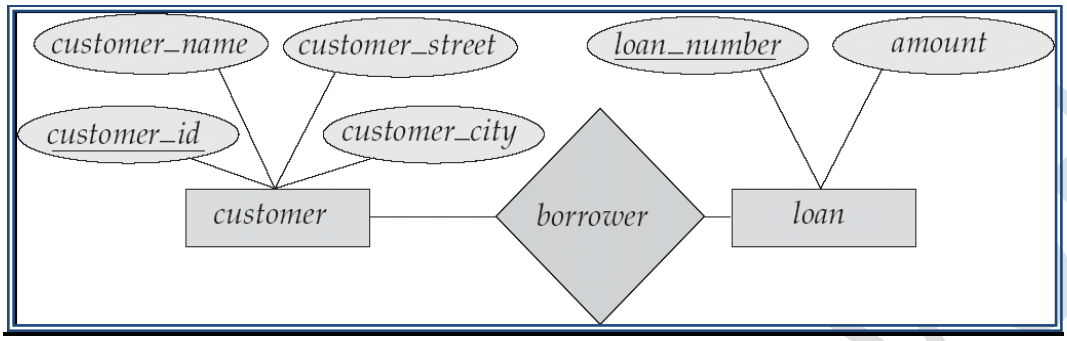
- A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.
- A **candidate key** of an entity set is a minimal super key
  - *Customer_id* is candidate key of *customer*
  - *account_number* is candidate key of *account*
- Although several candidate keys may exist, one of the candidate keys is selected to be the **primary key**.

Keys for Relationship Sets

- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
  - (*customer_id, account_number*) is the super key of *depositor*
  - *NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set.*

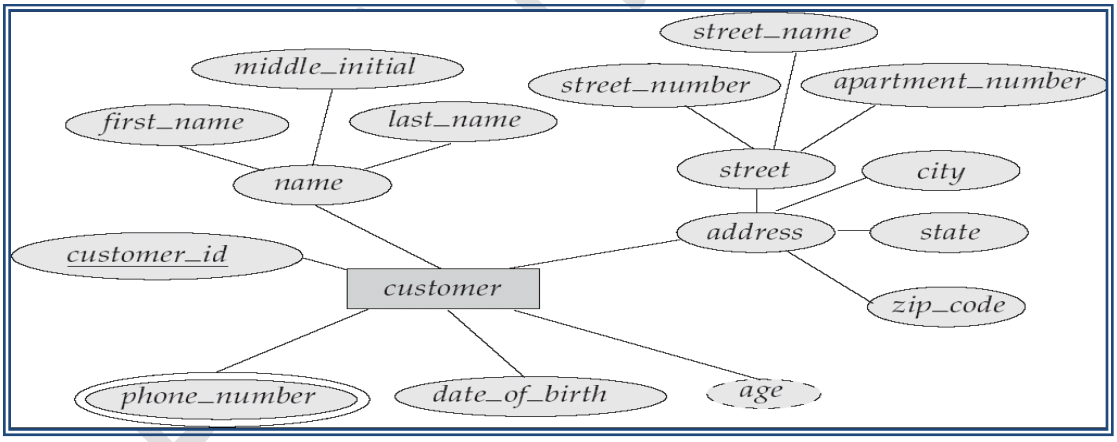
- Example: if we wish to track all access_dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalve attribute though

E-R Diagrams



- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Lines link attributes to entity sets and entity sets to relationship sets.
- Ellipses represent attributes
    - Double ellipses represent multivalve attributes.
    - Dashed ellipses denote derived attributes.
- Underline indicates primary key attributes

E-R Diagram With Composite, Multivalve, and Derived Attributes

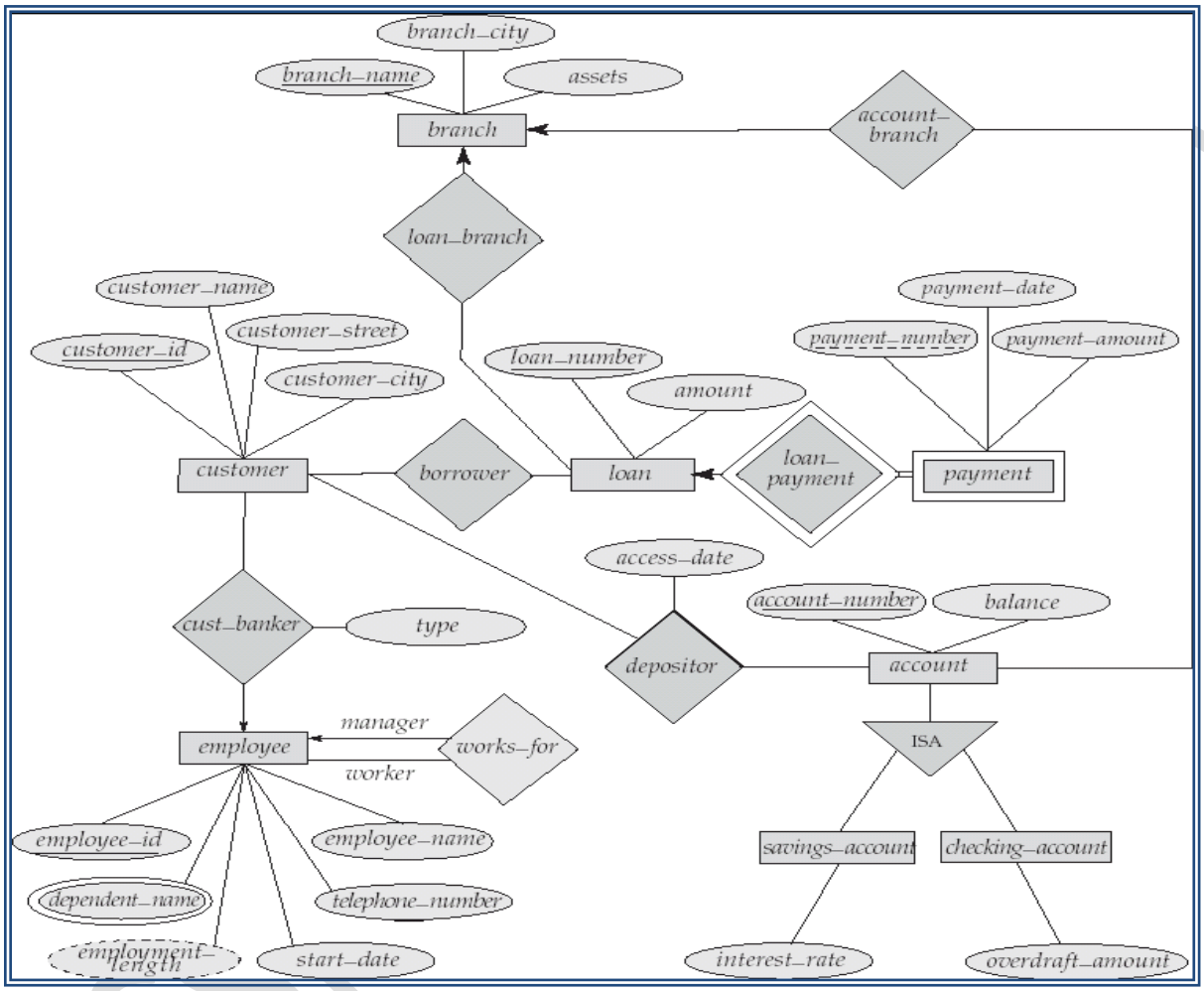


E-R Design Decisions

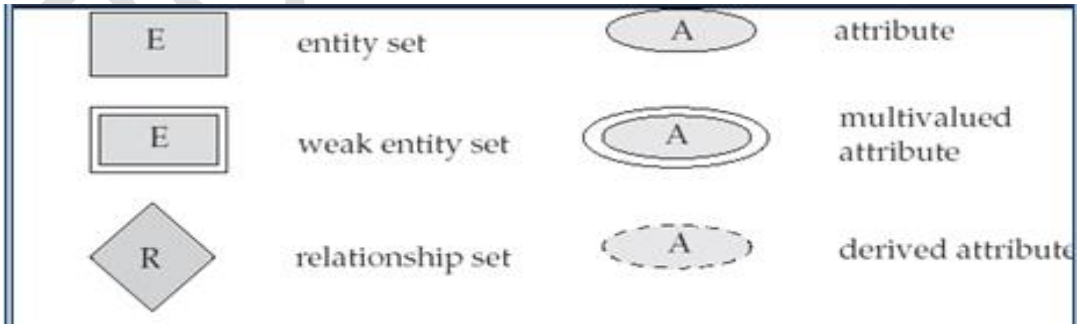
- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.

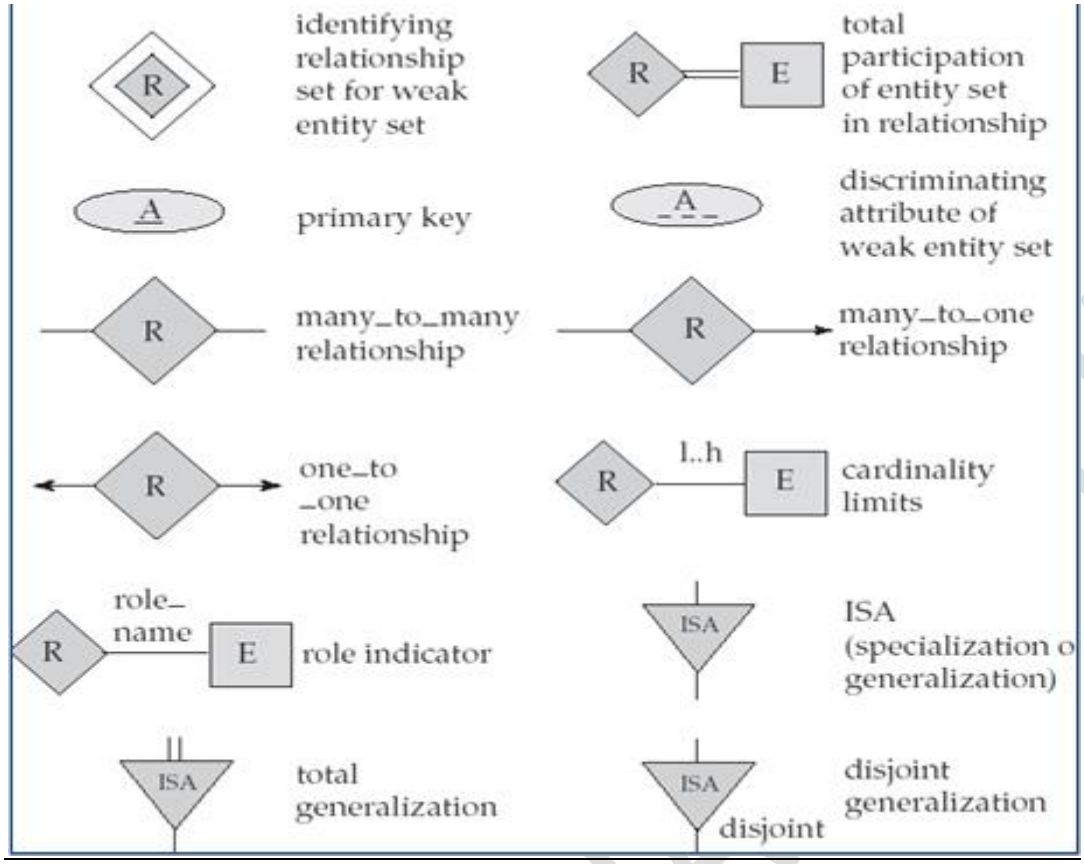
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.
-

E-R Diagram for a Banking Enterprise



Summary of Symbols Used in E-R Notation




identifying relationship set for weak entity set
R
total participation of entity set in relationship
R
E
A
primary key
_A_
discriminating attribute of weak entity set
R
many_to_many relationship
R
many_to_one relationship
R
one_to _one relationship
R
l..h
E
cardinality limits
role_ name
R
E
role indicator
ISA
ISA (specialization o generalization)
ISA
total generalization
ISA
disjoint
disjoint generalization

Unit V

2 Marks

1. **What is normalization?**
   - In relational database design, the process of organizing data to minimize redundancy.
   - Normalization usually involves dividing a database into two or more tables and defining relationships between the tables.
2. **What is primary and composite key?**
   - **Primary key:** A primary is a single column values used to uniquely identify a database record.
     - **E.g.:** Accno uniquely identify tuples in account relation.
   - **Composite key:** A composite key is a primary key composed of multiple columns used to identify a record uniquely.
     - Eg. Cname,accno are combined to uniquely tuples in depositor relation.
3. **Define Functional dependencies?**
   - Constraints on the set of legal relations.
   - Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
   - A functional dependency is a generalization of the notion of a *key*.
   - The functional dependency $\alpha \rightarrow \beta$

     Holds on *R* if and only if for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is,

     $$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$
4. **Define atomic domain and first normal form?**
   - Domain is atomic if its elements are considered to be indivisible units
     - Examples of non-atomic domains:
       - Set of names, composite attributes
       - Identification numbers like CS101 that can be broken up into parts
   - A relational schema R is in first normal form if the domains of all attributes of R are atomic
   - Non-atomic values complicate storage and encourage redundant (repeated) storage of data
   - Example: Set of accounts stored with each customer, and set of owners stored with each account
5. **What is Armstrong's axiom?**

The rules to find logically implied functional dependencies. By applying these rules repeatedly, we can find all of F+, given F. This collection of rules is called **Armstrong's axioms** in honor of the person who first proposed it.

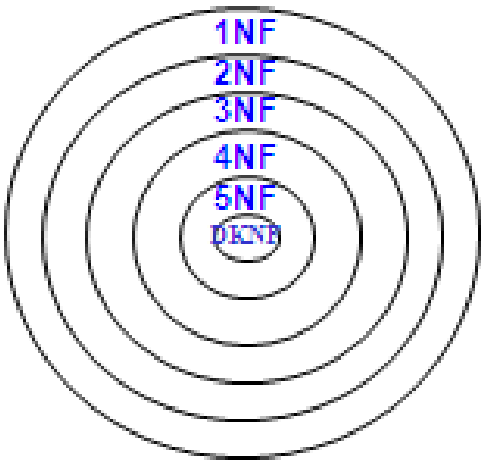
- **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.
- **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

6. What is Boyce-codd normal form?

- A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in F^+ of the form
  - $\alpha \rightarrow \beta$
- where $\alpha \subseteq R$ and $\beta \subseteq R$,at least one of the following holds:
  - $\alpha \rightarrow \beta$is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for *R*
- Example schema *not* in BCNF:
  - *bor_loan* = (*customer_id, loan_number, amount*)
- because *loan_number*→*amount* holds on *bor_loan* but *loan_number* is not a superkey

7. List out the various levels of normalizations

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)
- Domain Key Normal Form (DKNF)



5 Marks

8. Discuss briefly about the design goals of relational database design and multi-value dependencies.

- Goal for a relational database design is:
  - BCNF.

- Lossless join.
- Dependency preservation.

- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.
  - Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

9. Write short note on decomposition using functional dependencies with example.

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.
- Let *R* be a relation schema
- $\alpha \subseteq R$ *and* $\beta \subseteq R$
- The functional dependency
- $\alpha \rightarrow \beta$

  holds on*R* if and only if for any legal relations *r*(R), whenever any two tuples $t_1$and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is,
- $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$
- Example: Consider *r*(A,*B*) with the following instance of *r*.
- On this instance, $A \rightarrow B$ does NOT hold, but $B \rightarrow A$ does hold.
- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:
- bor_loan = (customer_id, loan_number, amount).

- We expect this functional dependency to hold:
- loan_number → amount
- but would not expect the following to hold:
- amount → customer_name

10. Explain the Use of Functional Dependencies

- We use functional dependencies to:
  - Test relations to see if they are legal under a given set of functional dependencies.
- If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* satisfies *F*.
  - specify constraints on the set of legal relations
- We say that *F* holds on *R* if all legal relations on *R* satisfy the set of functional dependencies *F*.
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *loan* may, by chance, satisfy

amount →*customer_name.*

- *A* functional dependency is trivial if it is satisfied by all instances of a relation
  - Examples*:*
    - *customer_name, loan_number* →*customer_name*
    - *customer_name* →*customer_name*

In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set *F* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - For example: If *A*→*B* and *B*→*C*, then we can infer that *A*→*C*
- The set of all functional dependencies logically implied by *F* is the *closure* of *F*.
- We denote the *closure* of *F* by F^+.
- F^+ is a superset of *F*.

10 Marks

11. What are the characteristic of Boyce-codd normal form?

- A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in F^+ of the form

- $\alpha \rightarrow \beta$

- where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R
- Example schema *not* in BCNF:
  bor_loan = (*customer_id, loan_number, amount*)
- because *loan_number*$\rightarrow$*amount* holds on *bor_loan* but *loan_number* is not a superkey

Decomposing a Schema into BCNF

- Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.
  We decompose R into:

  $(\alpha \cup \beta)$

  $(R - (\beta - \alpha))$
- In our example,
  - α = *loan_number*
  - β= *amount*

  and *bor_loan* is replaced by
  - $(\alpha \cup \beta)$ = (*loan_number, amount*)
  - $(R - (\beta - \alpha))$ = (*customer_id, loan_number*)

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.*
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database
  - *classes* (*course, teacher, book*)

- such that (*c*, *t*, *b*) ∈*classes* means that *t* is qualified to teach *c*, and *b* is a required textbook for *c*
- The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

course	*teacher*	*book*
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	Pete	Stallings

classes

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted

(database, Marilyn, DB Concepts)

(database, Marilyn, Ullman)

- Therefore, it is better to decompose *classes* into:

course	*teacher*
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	*book*
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

12. Discuss briefly about the design goals of relational database design and multi-value dependencies.

<u>Design Goals</u>

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation

- Redundancy due to use of 3NF

- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.
  - Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Multivalve Dependencies (MVDs)

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The *multivalve dependency*

  $\alpha \rightarrow\rightarrow \beta$

  holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

  $$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
  $$t_3[\beta] = t_1[\beta]$$
  $$t_3[R - \beta] = t_2[R - \beta]$$
  $$t_4[\beta] = t_2[\beta]$$
  $$t_4[R - \beta] = t_1[R - \beta]$$

Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

  Y, Z, W

- We say that $Y \rightarrow\rightarrow Z$ (Y multi-determines Z)

  if and only if for all possible relations r (R)

  $<y_1, z_1, w_1> \in r$ and $<y_1, z_2, w_2> \in r$

  then

  $<y_1, z_1, w_2> \in r$ and $<y_1, z_2, w_1> \in r$

- Note that since the behavior of Z and W are identical it follows that

  $Y \rightarrow\rightarrow Z$ if $Y \rightarrow\rightarrow W$

- In our example:

  course $\rightarrow\rightarrow$ teacher

  course $\rightarrow\rightarrow$ *book*

- The above formal definition is supposed to formalize the notion that given a particular value of *Y* (*course*) it has associated with it a set of values of *Z (teacher)* and a set of values of *W (book)*, and these two sets are in some sense independent of each other.
- Note:
    - If $Y \rightarrow Z$ then $Y \rightarrow\rightarrow Z$
    - Indeed we have (in above notation) $Z_1 = Z_2$

      The claim follows.

Use of Multivalve Dependencies

- We use multivalve dependencies in two ways:
    1. To test relations to determine whether they are legal under a given set of functional and multivalve dependencies
    2. To specify constraints on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalve dependencies.
- If a relation *r* fails to satisfy a given multivalve dependency, we can construct a relations *r′* that does satisfy the multivalve dependency by adding tuples to *r*.

13. Explain in detail about various normal forms used for Normalization.

1st Normal Form

A database is in first normal form if it satisfies the following conditions:

- Contains only atomic values
- There are no repeating groups

An atomic value is a value that cannot be divided. For example, in the table shown below, the values in the [Color] column in the first row can be divided into "red" and "green", hence [TABLE_PRODUCT] is not in 1NF.

1st Normal Form Example

How do we bring an un-normalized table into first normal form? Consider the following example:

TABLE_PRODUCT

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

This table is not in first normal form because the [Color] column can contain multiple values. For example, the first row includes values "red" and "green."

To bring this table to first normal form, we split the table into two tables and now we have the resulting tables:

TABLE_PRODUCT_PRICE

Product ID	Price
1	15.99
2	23.99
3	17.50
4	9.99
5	29.99

TABLE_PRODUCT_COLOR

Product ID	Color
1	red
1	green
2	yellow
3	green
4	yellow
4	blue
5	red

Now first normal form is satisfied, as the columns on each table all hold just one value.

2nd Normal form

A database is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-key attributes are fully functional dependent on the primary key

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.

Consider the following example:

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location]. In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key. Therefore, this table does not satisfy second normal form.

To bring this table to second normal form, we break the table into two tables, and now we have the following:

TABLE_PURCHASE

Customer ID	Store ID
1	1
1	3
2	1
3	2
4	3

TABLE_STORE

Store ID	Purchase Location
1	Los Angeles
2	New York
3	San Francisco

3rd Normal Form Definition

A database is in third normal form if it satisfies the following conditions:

- It is in second normal form
- There is no transitive functional dependency

By transitive functional dependency, we mean we have the following relationships in the table: A is functionally dependent on B, and B is functionally dependent on C. In this case, C is transitively dependent on A via B.

3rd Normal Form Example

Consider the following example:

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

In the table able, [Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type]. Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

To bring this table to third normal form, we split the table into two as follows:

TABLE_BOOK

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

Now all non-key attributes are fully functional dependent only on the primary key. In [TABLE_BOOK], both [Genre ID] and [Price] are only dependent on [Book ID]. In [TABLE_GENRE], [Genre Type] is only dependent on [Genre ID].

Boyce-Codd Normal Form (BCNF)

A relation is in Boyce-Codd Normal Form (BCNF) if every determinant is a candidate key.

A candidate key is a combination of attributes that can be uniquely used to identify a database record without any extraneous data. Each table may have one or more candidate keys. One of these candidate keys is selected as the table primary key.

Consider the following non-BCNF table whose functional dependencies follow the {AB → C, C → B} pattern:

Nearest Shops		
Person	**Shop Type**	**Nearest Shop**
Davidson	Optician	Eagle Eye
Davidson	Hairdresser	Snippets
Wright	Bookshop	Merlin Books
Fuller	Bakery	Doughy's
Fuller	Hairdresser	Sweeney Todd's
Fuller	Optician	Eagle Eye

The candidate keys of the table are:

- {Person, Shop Type}
- {Person, Nearest Shop}

Shop Near Person	
Person	**Shop**
Davidson	Eagle Eye
Davidson	Snippets
Wright	Merlin Books
Fuller	Doughy's
Fuller	Sweeney Todd's
Fuller	Eagle Eye

Shop	
Shop	**Shop Type**
Eagle Eye	Optician
Snippets	Hairdresser
Merlin Books	Bookshop
Doughy's	Bakery
Sweeney Todd's	Hairdresser

Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalve dependencies if for all multivalve dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema R

- If a relation is in 4NF it is in BCNF

Restriction of Multivalve Dependencies

- The restriction of D to R_i is the set D_i consisting of
  - All functional dependencies in D^+ that include only attributes of R_i
  - All multivalve dependencies of the form

    $\alpha \rightarrow\rightarrow (\beta \cap R_i)$

  where $\alpha \subseteq R_i$ and $\alpha \rightarrow\rightarrow \beta$ is in D^+

Example

- $R = (A, B, C, G, H, I)$

  $F = \{ A \rightarrow\rightarrow B$

  $B \rightarrow\rightarrow HI$

  $CG \rightarrow\rightarrow H \}$

- R is not in 4NF since $A \rightarrow\rightarrow B$ and A is not a superkey for R
- Decomposition

  a) $R_1 = (A, B)$ (R_1 is in 4NF)

  b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF)

  c) $R_3 = (C, G, H)$ (R_3 is in 4NF)

  d) $R_4 = (A, C, G, I)$ (R_4 is not in 4NF)

- Since $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$, $A \rightarrow\rightarrow HI$, $A \rightarrow\rightarrow I$

  e) $R_5 = (A, I)$ (R_5 is in 4NF)

  f) $R_6 = (A, C, G)$ (R_6 is in 4NF)

14. Explain in detail about database design process.

Overall Database Design Process

- We have assumed schema R is given
  - R could have been generated when converting E-R diagram to a set of tables.
  - R could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
  - Normalization breaks R into smaller relations.
  - R could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
    - Example: an *employee* entity with attributes *department_number* and *department_address*, and a functional dependency *department_number* $\rightarrow$ *department_address*
    - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

De-normalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *customer_name* along with *account_number* and *balance* requires join of *account* with *depositor*
- Alternative 1: Use de-normalized relation containing attributes of *account* as well as *depositor* with all above attributes
    - faster lookup
    - extra space and extra execution time for updates
    - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as

  account $\bowtie$ depositor
    - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors