Semester	**:**	III
Academic Year	**:**	2021
Course	:	**DATABASE MANAGEMENT SYSTEM**
Attendance	:	Minimum 75% failing which students may not be allowed to appear in the semester examinations

Disclaimer statement

"This course material booklet contains information compiled from variety of sources including standard text books and electronic resources for academic benefits of students to be used by them only as complementary to class room lectures. Citations of references to the text are made wherever possible. These notes are not meant for any commercial purpose and are solely meant for internal circulation."

Syllabus: DATABASE MANAGEMENT SYSTEM

Module – I

Introduction: Purpose of Database System; View of Data, Data Models, Database Languages, Transaction Management, Storage Management, Database Users Administrator, History of Database Systems.

Module – II

Database Design and Entity - Relational Model: Basic Concepts, Design issues Mapping, Constraints, Keys, E – R Diagram, Week Entity Sets, Extended E – R Features, Design of an E – R Database Schema, Deduction of an E – R Schema to Tables.

Module – III

Relational Model: Structure of Relational Database, Relational Algebra, Operation, Additional Operations, Calculus, Domain Relational Calculus, Tuple Relational, Query by Examples.

Module – IV

AQL & Other Relational Languages: Structures, Set Operations, Aggregate Functions, Null Values, Nested Sub – Queries, Derived Relations, Joined Relations, DDL, Other SQL features.

Module – V

Integrity Constraints: Domain Constraints, Referential Integrity, Assertions, Triggers & Functional Dependencies.

Module – VI

Relational Database Design: Pitfalls in Relational – Database Design, Functional Dependencies, Decomposition, Desirable Properties of Decomposition, Normalization (INF- DKNF), BCNF & Its Comparison with 3NF.

Module – VII

Query Processing: Measure of Query Cost, Evaluation of Expressions, Selection Operation.

Module – VIII

Transaction & Concurrency Control: Transaction Concepts & ACID Properties, Transaction States, Concurrent Executions, Serializability & Its Testing, Guarantee Serializability, Recoverability, Introduction to Concurrency Control, Locked Base Protocol & Deadlock Handling.

Text Book:

1. A.Silberschatz et.al - Database System Concepts, 5th Edn, Tata Mc-Graw Hill, New Delhi – 2000.

Reference Books:

1. Date C.J. - An Introduction to Database System, Pearson Education, New Delhi-2005
2. R.Elmasri, Fundamentals of Database Systems, Pearson Education, New Delhi,

COURSE PLAN

Department : **Computer Science**

Subject : **DATABASE MANAGEMENT SYSTEM**

Semester & branch : **III & BCA**

No. of periods hours/week : **4 Theory : yes Labs: yes**

Total No. of lectures : **40**

Recommended Course Books

Text Book:

1. A.Silberschatz et.al - Database System Concepts, 5th Edn, Tata Mc-Graw Hill, New Delhi – 2000.

Reference Books:

2. Date C.J. - An Introduction to Database System, Pearson Education, New
3. R.Elmasri, Fundamentals of Database Systems, Pearson Education, New Delhi,

Lecture No.	Topic(s) to be covered
1	Introduction: Purpose of Database System
2	View of Data,
3	Data Models
4	Data Models
5	Database Languages
6	Transaction Management

7	Storage Management
8	Database Users Administrator
9	History of Database Systems
10	Relational Model: Basic Concepts
11	Design issues Mapping, Constraints
12	Keys, E – R Diagram
13	Keys, E – R Diagram
14	Week Entity Sets, Extended E – R Features,
15	Week Entity Sets, Extended E – R Features,
16	Design of an E – R Database Schema
17	Deduction of an E – R Schema to Tables.
18	Deduction of an E – R Schema to Tables.
19	Integrity Constraints:Domain Constraints, Referential Integrity, Assertions, Triggers & Functional Dependencies . Relational Database Design: Pitfalls in Relational – Database Design, Functional Dependencies, Decomposition, Desirable Properties of Decomposition, Normalization (INF- DKNF), BCNF & Its Comparison with 3NF.
20	Assertions, Triggers
21	Functional Dependencies
22	Pitfalls in Relational – Database Design
23	Decomposition,
24	Desirable Properties of Decomposition,
25	Normalization
26	Normalization

27	BCNF & Its Comparison with 3NF.
28	BCNF & Its Comparison with 3NF.
29	Query Processing: Measure of Query Cost
30	Evaluation of Expressions
31	Evaluation of Expressions
32	Selection Operation.
33	Transaction Concepts & ACID Properties
34	TransactionStates, Concurrent Executions
35	Serializability & Its Testing, Guarantee Serializability,
36	Recoverability, Introduction to Concurrency Control,
37	Locked Base Protocol & Deadlock Handling.
38	SQL & Other Relational Languages
39	Relational Model
40	Relational Model

Test Coverage :

Test1:

Introduction: Purpose of Database System; View of Data, Data Models, Database Languages, Transaction Management, Storage Management, Database Users Administrator, History of Database Systems.

Database Design and Entity - Relational Model: Basic Concepts, Design issues Mapping, Constraints, Keys, E – R Diagram, Week Entity Sets, Extended E – R Features, Design of an E – R Database Schema, Deduction of an E – R Schema to Tables.

Test2:

Integrity Constraints: Domain Constraints, Referential Integrity, Assertions, Triggers & Functional Dependencies .

Relational Database Design: Pitfalls in Relational – Database Design, Functional Dependencies, Decomposition, Desirable Properties of Decomposition, Normalization (INF- DKNF), BCNF & Its Comparison with 3NF.

Outcome /Benefits of the course:

One can understand the internal details of Relational Algebra. You can write a efficient query based on many algorithms. This course will help students in developing software and give them the solution of how to avoid deadlock in real life problem like banking system, Airline Management System etc., concepts for transaction processing and the operations relevant to transaction processing, types of failures that may occur during transaction execution, concurrency control, distributed databases and centralized databases. Design and build a relational database for a small business application form given user requirements

This course presents problem-solving logic and skills for analyzing and developing a database. Topics covered will include database design, administration and application development. This course is designed to give the student the understanding necessary to work efficiently within the database environment. Optimize a database for commercial operation

Unit I

Introduction: Purpose of Database System; View of Data, Data Models, Database Languages, Transaction Management, Storage Management, Database Users Administrator, History of Database Systems.

DATABASE SYSTEM CONCEPTS AND ARCHITECTURE

Data Base Management System (DBMS):

DBMS (Database Management System) consists of a collection of interrelated data and a collection of programs to access that data .

Database:

Collection of data

Why Database (Needs and Benefits) – An enterprise chooses to store its operational data in an integrated data base because, broadly, a DBS provides the enterprise with centralized control of its operational data as data is one of its most valuable assets.

Benefits of database approach:

- **Redundancy can be reduced** : If DBA is aware of data requirements for the applications, it could be controlled to some extent but not completely eliminated. Redundancy results into wastage of storage space.
- **Inconsistency can be avoided**: Ex. Employee EMP3 works in DEP9(, is represented by two distinct entries in the DB, and that the system is not aware of this duplication; in other words, the redundancy is not controlled. Then, there will be some occasions on which the two entries will not agree, that is, when one and only one has been updated. At such times the DB is said to be inconsistent. A DB that is in an inconsistent state is capable of supplying incorrect or conflicting information.
- **The Data can be Shared**: It might be possible to satisfy the data requirements of new applications without having to create any additional stored data.
- **Standards can be enforced**: With central control of the DB, the DBA can ensure that all applicable standards are observed in the representation of data. Applicable standards might include any or all of the following: Corporate, installation, departmental, industry, national and international standards.
  Standardizing data representation is particularly desirable as an aid to data interchange, or migration of data between systems.
- **Security restrictions can be applied :** DBA 1) define security rules to be checked wherever access is attempted to sensitive data
  2) can ensure that the only means of access data to the DB is thru the proper channels.

- **Integrity can be maintained**: Problem of ensuring that the data in the database is correct and accurate. Due to inconsistency and lack of integrity redundancy exists in the stored data, DB may become incorrect.
- **Conflicting requirements can be balanced**: Knowing the overall requirements of the enterprise- the DBA can so structure the system as to provide an overall service that is 'best for the enterprise'.

Data Independence:

Data Independence can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. For example:

method of representation of alphanumeric data (e.g., changing date format to avoid Y2000 problem)

method of representation of numeric data (e.g., integer vs. long integer or floating-point)

units (e.g., metric vs. furlongs)

We can Define two types of data Independence:

Logical Data Independence:

Capacity to change the conceptual schema with out having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item) , or to reduce the database (by removing a record type or data items).

Physical Data Independence:

Capacity to change the Internal schema without having to change the conceptual or external schema. For example , by creating additional access structures –to improve the performance of retrieval or update.

Advantages of using a DBMS

a) Reduced data redundancy.
b) Reduced updating errors and increased consistency.
c) Greater data integrity and independence from applications programs.
d) Improved data access to users through use of host and query languages.
e) Improved data security.
f) Reduced data entry, storage, and retrieval costs.
g) Facilitated development of new applications program

Disadvantages of using a DBMS

a) Database systems are complex, difficult, and time-consuming to design.

b) Substantial hardware and software start-up costs.

c) Damage to database affects virtually all applications programs.

d) Extensive conversion costs in moving form a file-based system to a database system.

e) Initial training required for all programmers and users

Sr. No	File Processing System	Database Management System
1	A file-processing system only coordinates physical access to the data	A database coordinates the physical andlogical access to the data.
2	A file-processing system introduces the amount of data duplication.	A DBMS reduces the amount of dataduplication
3	A file-processing system only allows predeterminedaccess to data (by specific compiled programs).	A DBMS is designed to allow flexibility inwhat queries give access to the data.
4	A file processing system is much more restrictive in simultaneous data access	A DBMS is designed to coordinate andpermit multiple users to access data atthe same time
5	A file processing system does not support searching and theimplementation of right management	A DBMS supports improvement of searching and the implementation ofright management.
6	There is no way to restrict unauthorized access in a file processing system.	A DBMS restricts unauthorized access
7	There is no way to recover a lost file in file-processing system.	A DBMS supports backup and recoveryfrom system crashes
8	There is no way to enforces integrity constraints in a file processing system.	A DBMS enforces integrity constraints

View of Data :

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

Data Abstraction-

Developers hide the complexity from users thru several levels of abstraction, to simplify user's interactions with the system:

- **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 describes what data are stored in the DB, and what relationships exist among those data. DBA, who must decide what information to keep in the database, use the logical level of abstraction.
- **View Level-** The highest level of abstraction describes part of the entire database.

Ex. A banking enterprise may have record types

- Account, with fields account-number and balance
- Employee, with fields employee-name and salary

At the Physical level, a customer, account, or employee can be described as a block of consecutive storage locations (words and bytes). The language compiler hides this level of detail from programmers. DBA may be aware of certain details of the physical organization of the data.

At the logical level, each record is described by a type definition and interrelationship of these records types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, DBA usually work at this level of abstraction.

Finally, **at the view level**, computer users see a set of application programs that hide details of the data types. Similarly, at the view level, several views of the database are defined, and db users see these views. In hiding details of the logical level of the db, the views also provide a security mechanism to prevent users from accessing certain parts of the db. E.g., tellers in a bank see only the part of the database that has information on customer accounts; they cannot access information about salaries of employees.

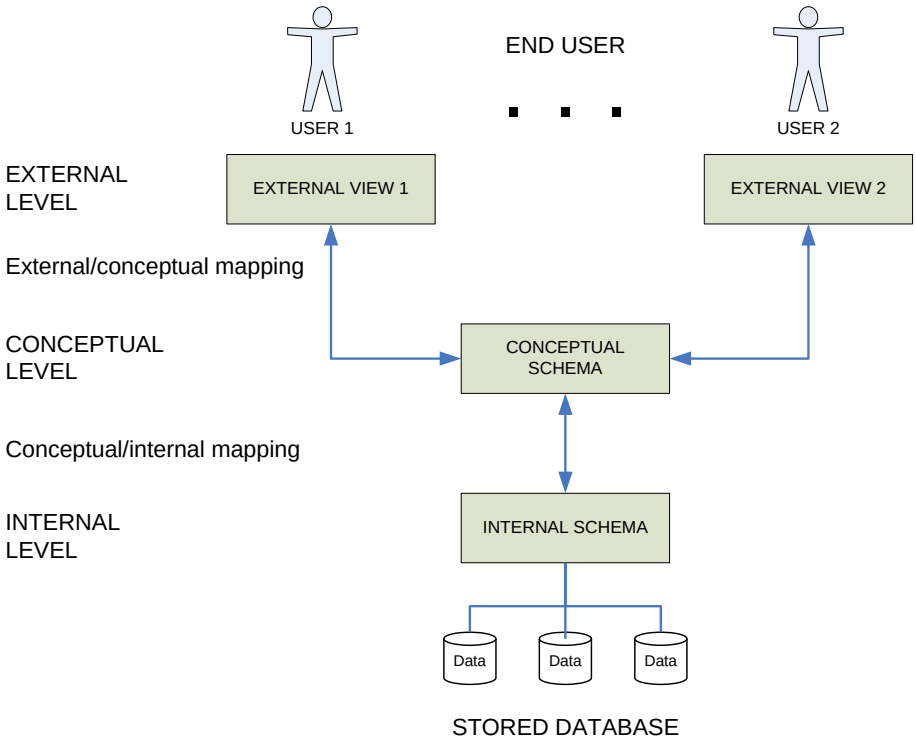
An Architecture for a Database System / Phases of Database Design

Three levels- Internal, Conceptual and External Levels

- The Internal Level is the closest to Physical storage- i.e., it is the one concerned with the way the data is physically stored; Gives Storage View(Representing the total database as physically stored)
- The External level is the one closest to the users- i.e., it is the one concerned with the way the data is viewed by individual users; and Gives individual user view
- The Conceptual level is a "level of indirection" between the other two. Depicts Community user

An Example of three levels

<table>
<tr><td><u>EXTERNAL (PL/I)</u>
DCL 1 EMPP,
2 EMP# CHAR(6),
2 SAL FIXED BIN(31);</td><td><u>EXTERNAL (COBOL)</u>
01 EMPC
02 EMPNO PIC X(6).
02 DEPTNO PIC X(4).</td></tr>
<tr><td colspan="2"><u>CONCEPTUAL</u>
EMPLOYEE
EMPLOYEE_NUMBER CHARACTER (6)
DEPARTMENT_NUMBER CHARACTER (4)
SALARY NUMERIC (5)</td></tr>
<tr><td colspan="2"><u>INTERNAL</u>
STORED_EMP LENGTH=20
PREFIX TYPE=BYTE(6), OFFSET=0
EMP# TYPE=BYTE(6), OFFSET=6, INDEX=EMPX
DEPT# TYPE=BYTE(4), OFFSET=12
PAY TYPE=FULLWORD, OFFSET=16</td></tr>
</table>



Instances and Schemas – DB change over time as information is inserted and deleted.

Instance:

The collection of information stored in the database at a particular moment is called an **instance** of the database.

Schema:

The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

Example. The values of the variables in a program at a point in time correspond to an instance of database schema.

DBS have several schemas, partitioned acc. to the level of abstraction. The **physical schema** describes the db design at the physical level, while the **logical schema** describes the db design at the logical level. A db may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the db.

Database Languages- A DBS provides a **data definition language (DDL)** to specify the database schema and a data manipulation language **(DML)** to express database queries and updates and data control language **(DCL).**

a) Data definition language – This is the means by which the content and format of data to be stored is described and structure of the db is defined, including relationships between records and indexing strategies. Often known as a schema.DDL is essentially the link between the logical and physical views of the db.

Example: in SQL

create table *account* (*account-number***char**(6), *balance***integer**)

Primary Functions of DDL

- Describe the schema and subschema
- Describe the fields in each record and the record's logical name
- Describe the data type and name of each field
- Indicates the keys on the record
- Provide the data security restrictions
- Provide for logical and physical data independence
- Provide means of associating related data

b) Data Manipulation Language- Data manipulation is

- The retrieval of information stored in the db
- The insertion of new information into the db
- The deletion of information from the db
- The modification of information stored in the db

A DML is a language that enables users to access or manipulate data as organized by the appropriate data model. They are basically 2 types:

- **Procedural DMLs** require a user to specify what data are needed and how to get those data. Ex. C Language Commands, c++, Java, Basic, fortran, cobol, Pascal
- **Non Procedural DMLs** require a user to specify what data are needed without specifying how to get those data.Example Lisp,Prolog

Procedural DML and Non Procedural DML Verbs are : Delete , Sort, Insert, Display, Add, select etc.

c) Data Control Language: Used for controlling data and access to the databases.

Primary Functions of DCL

- Aid the physical administration of the db such as dumping, logging, recovery, reorganization, db initialization, export and import of data etc.
- Help the DBA and system designer to coordinate and keep track of the data on the db such as DD.
- DCL commands: rollback, grant, alter, revoke etc.

Database Administrator – A person who has central control over the data and programs (System) is called a database administrator (DBA).

Functions of DBA

- Schema definition: The DBA creates the original db schema by executing a set of data definition statements in the DDL.
- Storage structure and access-method definition and strategy
- Schema and physical-organization modification: DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- Granting of authorization for data access: To prevent unauthorized access to the data in db DBA grants different types of authorization to db users.
- Routine maintenance
    - Periodic backup either on tapes or onto remote servers, to prevent loss of data in case of disasters.
    - Database performance tuning and optimization to ensure that enough free disk space is available for normal operations, and upgrading disk space as required.
    - Monitoring jobs running on the db and ensuring that performance is not degraded by very expensive tasks submitted by some users.

Database Users

- **Naive** Users who interact with the system by invoking one of the application programs that have been written previously.For example , a bank teller who needs to transfer $50 from account A to accout B invokes a program called transfer.
- **Application programmers** are computer professional who write application programs.
- **Sophisticated** users interact with the system without writing programs. For example analyst can see total sales by region (north,south,East, west) , or by product , or by product and region both.
- **Specialized users** are users who write specialized database application. Applications are computer aided design systems, knowledge base and expert system.

DATA MODELS

1) Entity-Relational model
2) Relational model
3) Network model
4) Hierarchical model

Entity Relational Model

- Primarily a database design tool.
- Complements the relational data model concepts
- Represented in an entity relationship diagram (ERD)
- Based on entities, attributes, and relationships

Advantages of E.R. Model

- Exceptional conceptual simplicity
- Visual representation
- Effective communication tool
- Integrated with the relational data model

Disadvantages of E.R. Model

- Limited constraint representation
- Limited relationship representation
- No data manipulation language
- Loss of information content

The Relational Model

- Consists of tables; links among entities are maintained with foreign keys
- Advantages of relational databases
  - Same advantages of a network database without the complications
  - Easier to conceptualize and maintain
  - Virtually all DBMSs offered for microcomputers accommodate the relational model

The Network Model

- Allows a record to be linked to more than one parent
- Supports many-to-many relationships
- Advantage of the network model
  - Reduced data redundancy
  -
- Disadvantages of the network model
  - Complicated to build and difficult to maintain
  - Difficult to maintain and navigate

The Hierarchical Model

- Records are related hierarchically—each category is a subcategory of the next level up

Advantages

It Promotes data security
It Promotes data independence
It Promotes data integrity(parent/child relationship)
Useful for large databases
Useful when users require a lot of transactions which are fixed over time
Suitable for large storage media

- **Disadvantages of hierarchical databases**

  - To retrieve a record, a user must start at the root and navigate the hierarchy.
  - If a link is broken, the entire branch is lost.
    - Requires considerable data redundancy

Six Major steps that need to be taken in setting up a database for a particular enterprise.

1) Define the high level requirements of the enterprise (System requirements Specification)
2) Define a model containing all appropriate types of data and data relationships.
3) Define the integrity constraints on the data.
4) Define the physical level
5) For each known problem to be solved on regular basis
6) Create/ initialize the database

Responsibilities of a database manager/ Storage Manager

A storage manager is a program module that provides the interface between the low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with file manager. The storage manager translates the various DML statements into low level file system commands. Thus Storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

- Authorization and integrity manager
- Transaction manager (which ensure database remains in a consistent state)
- File manager (allocation of space on disk)
- Buffer manager (fetching data from disk into main memory, and deciding what data to cache in main memory)

The Storage manager implements several data structure:

- Data files
- Data dictionary
- Indices

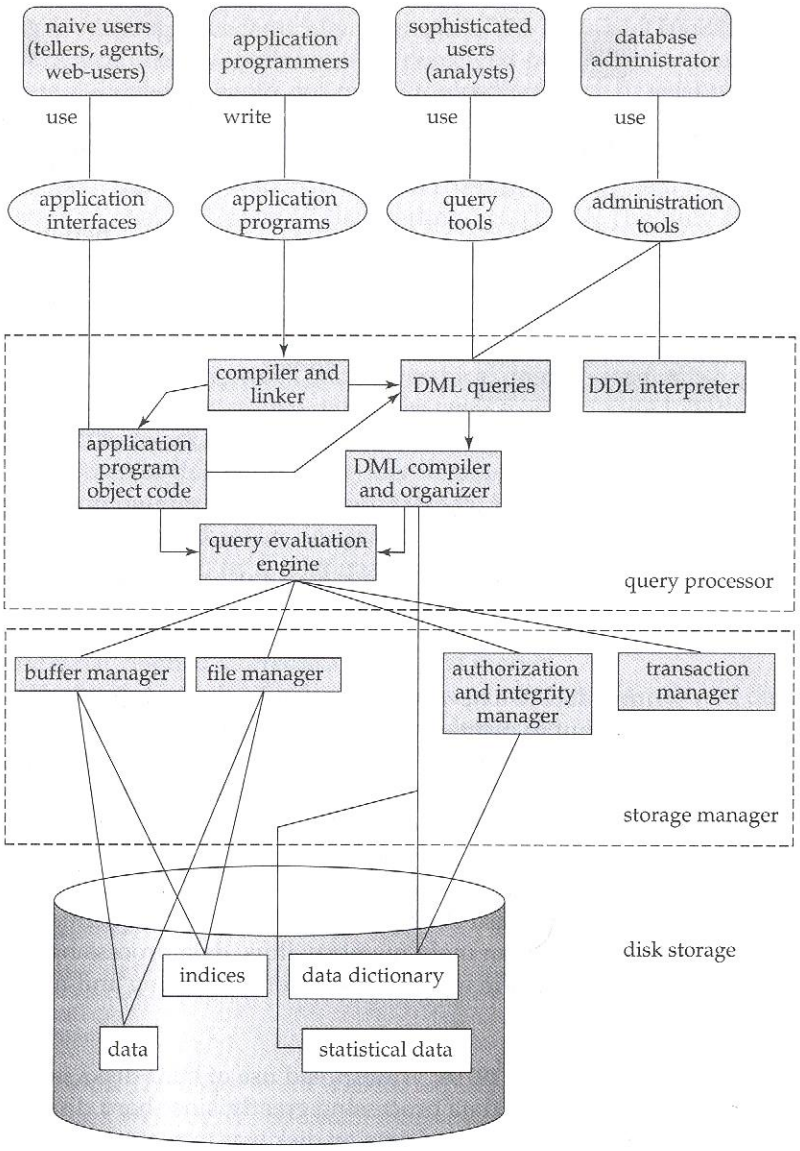


Figure 1.4 System structure.

History of Database Systems

1950s and early 1960s:

- Data processing using magnetic tapes for storage
  - Tapes provide only sequential access
- Punched cards for input

Late 1960s and 1970s:

- Hard disks allow direct access to data
- Network and hierarchical data models in widespread use
- Ted Codd defines the relational data model
  - Would win the ACM Turing Award for this work
  - IBM Research begins System R prototype
  - UC Berkeley begins Ingres prototype
- High-performance (for the era) transaction processing

1980s:

- Research relational prototypes evolve into commercial systems
  - SQL becomes industrial standard
- Parallel and distributed database systems
- Object-oriented database systems

1990s:

- Large decision support and data-mining applications
- Large multi-terabyte data warehouses
- Emergence of Web commerce

2000s:

- XML and XQuery standards
- Automated database administration

Unit II

Database Design and Entity - Relational Model: Basic Concepts, Design issues Mapping, Constraints, Keys, E – R Diagram, Week Entity Sets, Extended E – R Features, Design of an E – R Database Schema, Deduction of an E – R Schema to Tables.

DATA MODEL:

Collection of concepts that can be used to describe the structure if a database.

Use of High – Level Conceptual Data Models

ER Data model:

ER data model is a high level – data model. ER data model perceives the real world as consisting of basic objects, called entities and relationship among them.

Entities and Entity Sets

- An **entity** is an object that exists and is distinguishable from other objects. For instance, John Harris with S.I.N. 890-12-3456 is an entity, as he can be uniquely identified as one particular person in the universe.
- An entity may be **concrete** (a person or a book, for example) or **abstract** (like a holiday or a concept).
- An **entity set** is a set of entities of the same type (e.g., all persons having an account at a bank).
- Entity sets **need not be disjoint**. For example, the entity set *employee* (all employees of a bank) and the entity set *customer* (all customers of the bank) may have members in common.
- An entity is represented by a set of **attributes**.
    - E.g. name, S.I.N., street, city for ``customer" entity.
    - The **domain** of the attribute is the set of permitted values (e.g. the telephone number must be seven positive integers).
- Formally, an attribute is a **function** which maps an entity set into a domain.
    - Every entity is described by a set of (attribute, data value) pairs.
    - There is one pair for each attribute of the entity set.
    - E.g. a particular *customer* entity is described by the set {(name, Harris), (S.I.N., 890-123-456), (street, North), (city, Georgetown)}.

An analogy can be made with the programming language notion of type definition.

- The concept of an **entity set** corresponds to the programming language **type definition**.
- A variable of a given type has a particular value at a point in time.
- Thus, a programming language variable corresponds to an **entity** in the E-R model.

We will be dealing with five entity sets in this section:

- *branch*, the set of all branches of a particular bank. Each branch is described by the attributes *branch-name*, *branch-city* and *assets*.
- *customer*, the set of all people having an account at the bank. Attributes are *customer-name*, *S.I.N.*, *street* and *customer-city*.

- *employee*, with attributes *employee-name* and *phone-number*.
- *account*, the set of all accounts created and maintained in the bank. Attributes are *account-number* and *balance*.
- *transaction*, the set of all account transactions executed in the bank. Attributes are *transaction-number*, *date* and *amount*.

Relationships & Relationship Sets

A **relationship** is an association between several entities.

A **relationship set** is a set of relationships of the same type.

Formally it is a mathematical relation on $n \geq 2$ (possibly non-distinct) sets.

> A *relationship* set is a mathematical relation among $n \geq 2$ entities, each taken from entity sets
>
> $$\{(e1, e2, \ldots en) \mid e1 \in E1, e2 \in E2, \ldots, en \in En\}$$

where ($e1$, $e2$, …, en) is a relationship

Entity Sets *customer* and *loan*

Customer

Cust-id	Name	Street	City
321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	MAin	Harrison

Loan

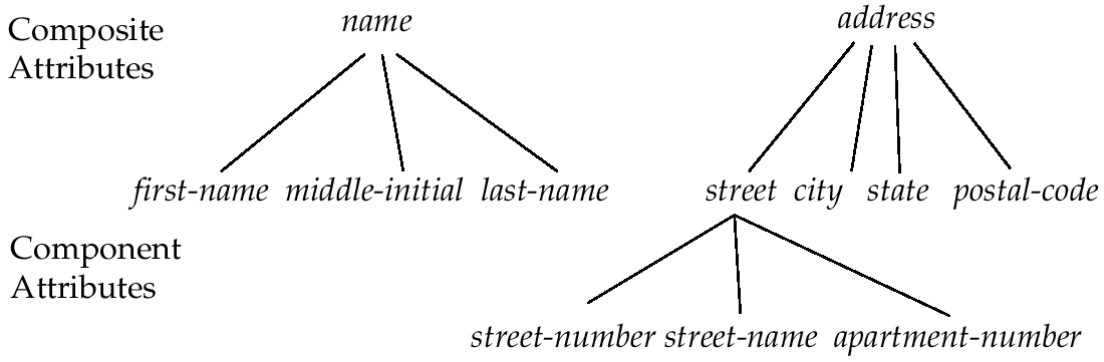
Loan-no	Amount
L-17	1000
L-23	2000
L-15	1500

Attributes

Consider the entity set *employee* with attributes *employee-name* and *phone-number*.

- We could argue that the phone be treated as an entity itself, with attributes *phone-number* and *location*.
- Then we have two entity sets, and the relationship set *EmpPhn* defining the association between employees and their phones.
- This new definition allows employees to have several (or zero) phones.
- New definition may more accurately reflect the real world.
- We cannot extend this argument easily to making *employee-name* an entity.

The question of what constitutes an entity and what constitutes an attribute depends mainly on the structure of the real world situation being modeled, and the semantics associated with the attribute in question.

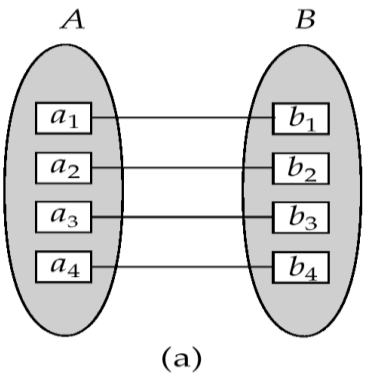


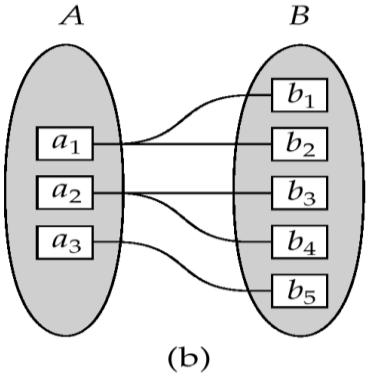
Mapping Constraints

An E-R scheme may define certain constraints to which the contents of a database must conform.

- **Mapping Cardinalities:** express the number of entities to which another entity can be associated via a relationship. For binary relationship sets between entity sets A and B, the mapping cardinality must be one of:
    1. **One-to-one**: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.
    2. **One-to-many**: An entity in A is associated with any number in B. An entity in B is associated with at most one entity in A.
    3. **Many-to-one**: An entity in A is associated with at most one entity in B. An entity in B is associated with any number in A.
    4. **Many-to-many**: Entities in A and B are associated with any number from each other.

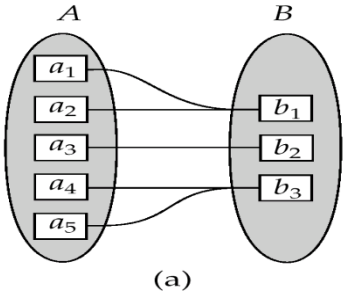
    The appropriate mapping cardinality for a particular relationship set depends on the real world being modeled. (Think about the *CustAcct* relationship...)

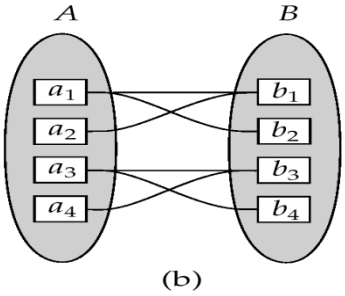




One to one

One to many





Many to one

Many to many

- **Existence Dependencies:** if the existence of entity X depends on the existence of entity Y, then X is said to be **existence dependent** on Y. (Or we say that Y is the **dominant** entity and X is the **subordinate** entity.)

  For example,

  - Consider *account* and *transaction* entity sets, and a relationship *log* between them.
  - This is one-to-many from account to transaction.
  - If an *account* entity is deleted, its associated *transaction* entities must also be deleted.
  - Thus *account* is dominant and *transaction* is subordinate.

Keys

Differences between entities must be expressed in terms of attributes.

- A **superkey** is a set of one or more attributes which, taken collectively, allow us to identify uniquely an entity in the entity set.
- For example, in the entity set *customer*, *customer-name* and *S.I.N.* is a superkey.
- Note that *customer-name* alone is not, as two customers could have the same name.
- A superkey may contain extraneous attributes, and we are often interested in the smallest superkey. A superkey for which no subset is a superkey is called a **candidate key**.
- In the example above, *S.I.N.* is a candidate key, as it is minimal, and uniquely identifies a customer entity.
- A **primary key** is a candidate key (there may be more than one) chosen by the DB designer to identify entities in an entity set.

An entity set that does not possess sufficient attributes to form a primary key is called a **weak entity set.** One that does have a primary key is called a **strong entity set**.

For example,

- The entity set *transaction* has attributes *transaction-number*, *date* and *amount*.
- Different transactions on different accounts could share the same number.
- These are not sufficient to form a primary key (uniquely identify a transaction).
- Thus *transaction* is a weak entity set.

Relationship Sets

A relationship is an association among several entities

Example:

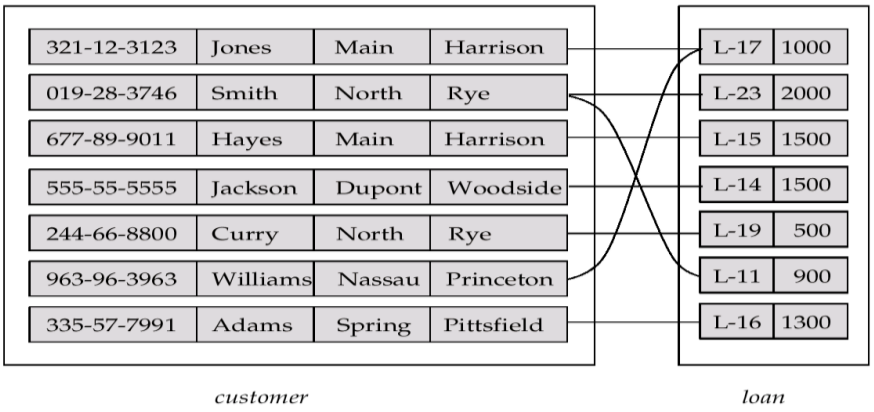
Hayes	depositor	A-102
customer entity	relationship set	account entity

A relationship set is a mathematical relation among $n \geq 2$ entities, each taken from entity sets$\{(e1, e2, \ldots en) \mid e1 \in E1, e2 \in E2, \ldots, en \in En\}$

where (e1, e2, …, en) is a relationship

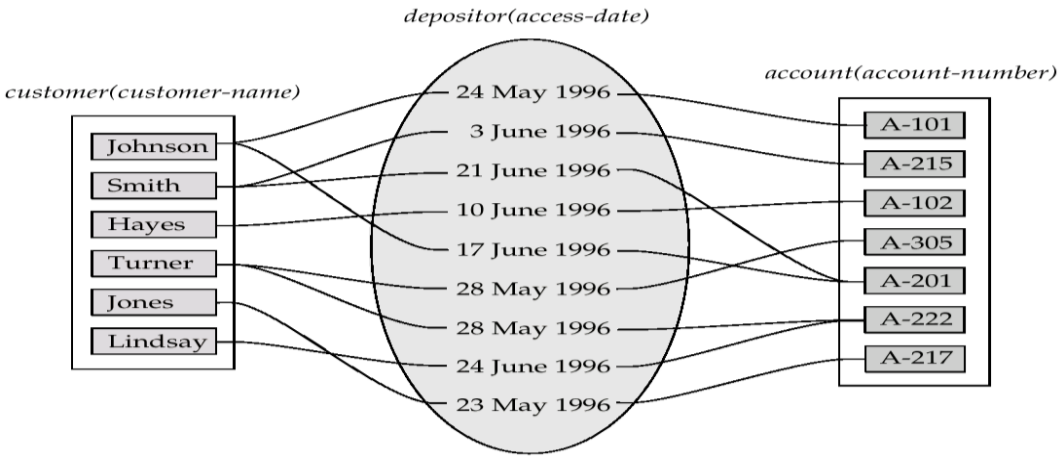
Example: (Hayes, A-102) $\in$ depositor

Relationship Set *borrower*



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
E.g. 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.

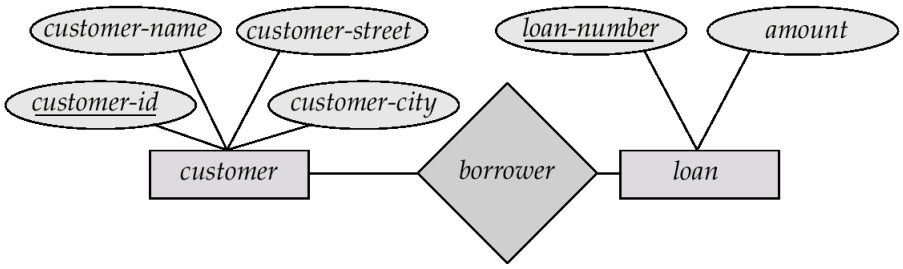
ER DIAGRAMS

The Entity Relationship Diagram

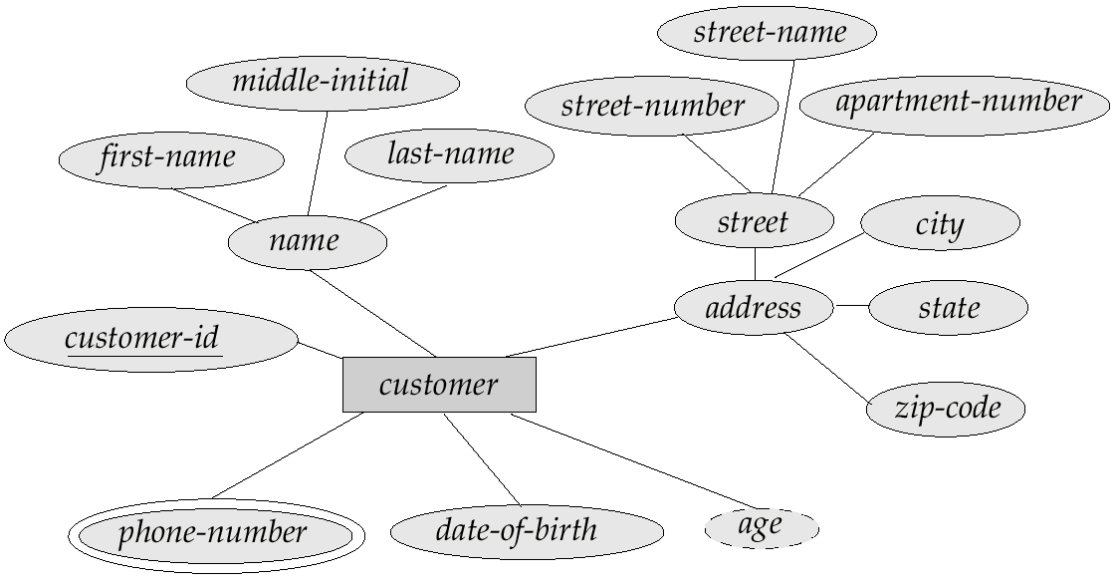
We can express the overall logical structure of a database **graphically** with an E-R diagram.

Its components are:

- **rectangles** representing entity sets.
- **ellipses** representing attributes.
- **diamonds** representing relationship sets.
- **lines** linking attributes to entity sets and entity sets to relationship sets.
- **Double Ellipse**, which represent multivalued attribute
- **Double Lines** , which indicate total participation of an entity in a relationship set
- **Double rectangle** , which represent week entity sets.



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



a)

Simple Attributes: The attribute that cannot be further divided into smaller parts and represents the basic meaning is called a simple attribute. e.g. The "First name", "Last Name" attributes of a EMPLOYEE entity represent a simple attribute.

Composite Attributes: The attributes that can be further divided into smaller units and each individual unit contains a specific meaning. For example, an attribute **name** of an entity set EMPLOYEE can be sub-divided into **First-name**, **Middle-initial**, and **Last-name**.

b)

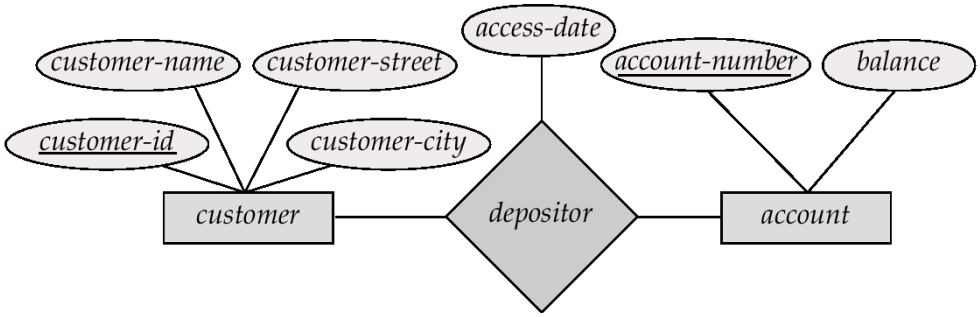
Single Valued Attribute: The attributes having single value for a particular entity is called as single-valued attribute. e.g. **age** is a single valued attribute of a EMPLOYEE entity.

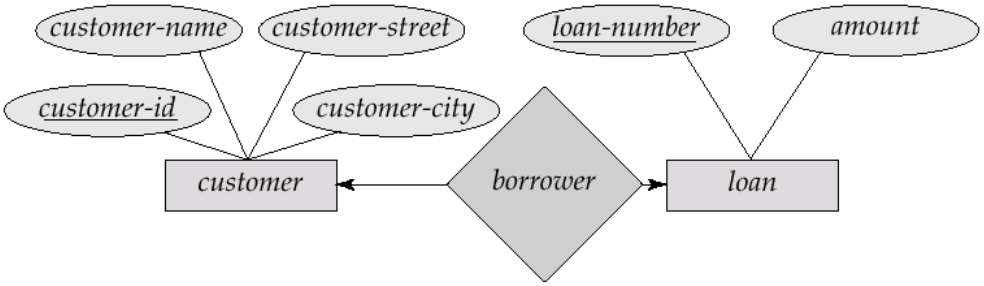
Multi-valued Attributes: Attributes that have more than one values for a particular entity is called a multi-valued attribute. Different entities may have different number of values for these kind of attributes. For multi-valued attributes we must also specify the minimum and maximum number of values that can be attached. e.g. **phone-number** for a EMPLOYEE entity is a multi-valued attribute.

c) Derived Attributes: The attributes that are not stored directly but can be derived from stored attributes are called derived attributes. e.g. **total-salary** of an entity EMPLOYEE can be calculated from **basic-salary** attribute.

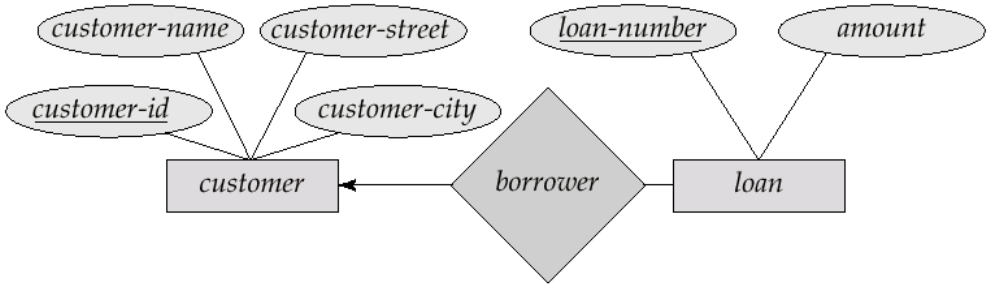
d) Null Value: An attribute takes **null value** when an entity does not have a value for it. The **null value** indicate "not applicable"- that is, that the value does not exist for the entity. For example, one may have no middle name. Null can also designate that an attribute value is unknown. An unknown value may be either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists).

Relationship Sets with Attributes

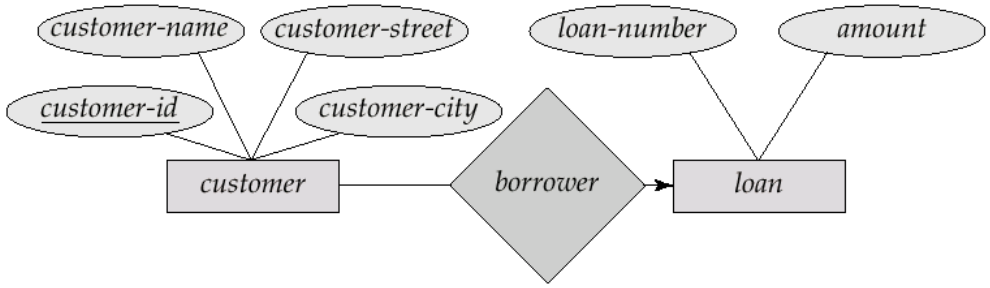




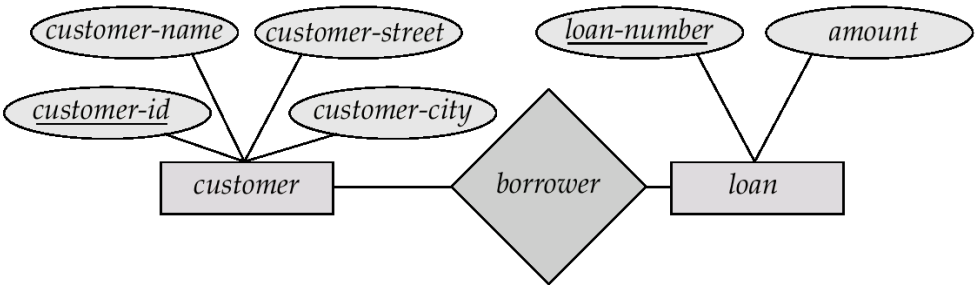
One to One



One to many

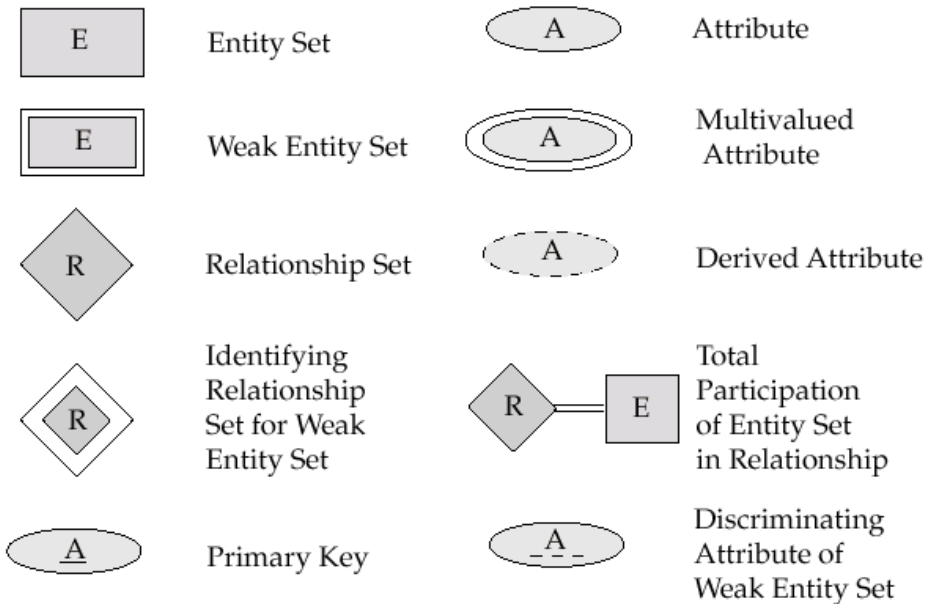


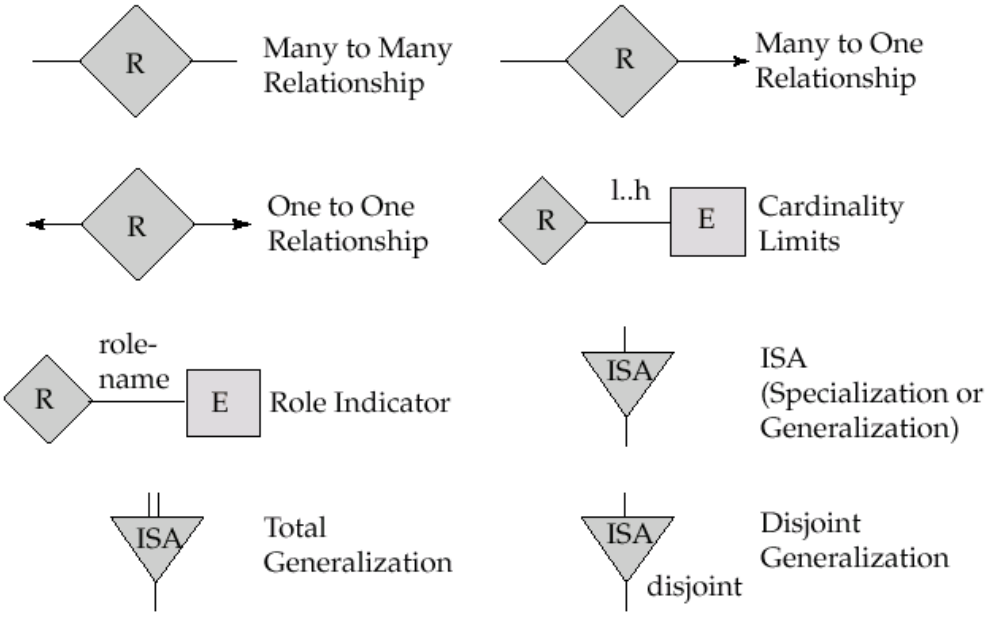
Many to one



May to many

Summary of Symbols Used in E-R Notation





- **Extended E-R diagrams** allowing more details/constraints in the real world to be recorded.
    - Composite Attributes.
    - Derived Attributes.
    - Subclasses and Superclasses.
    - Generalization and Specialization.

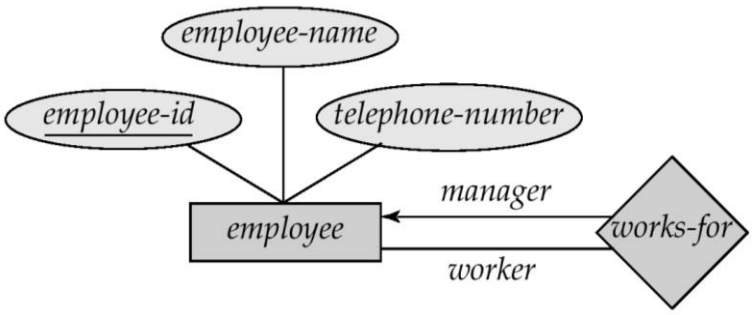
Roles in E-R Diagrams

The function that an entity plays in a relationship is called its **role**. Roles are normally explicit and not specified.

They are useful when the meaning of a relationship set needs clarification.

For example, the entity sets of a relationship may not be distinct. The relationship *works-for* might be ordered pairs of *employees* (first is manager, second is worker).

In the E-R diagram, this can be shown by labelling the lines connecting entities (rectangles) to relationships (diamonds).



E-R diagram with role indicators

Participation of an Entity Set in a Relationship Set

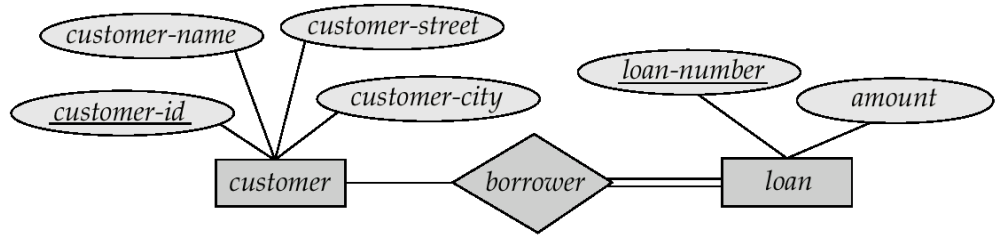
Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

E.g. participation of *loan* in *borrower* is total

every loan must have a customer associated to it via borrower

Partial participation: some entities may not participate in any relationship in the relationship set

E.g. participation of *customer* in *borrower* is partial



Weak Entity Sets in E-R Diagrams

An entity set that does not have a primary key is referred to as a *weak entity set*.

The existence of a weak entity set depends on the existence of an*identifying entity set*

It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set

Identifying relationship depicted using a double diamond

The *discriminator (or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.

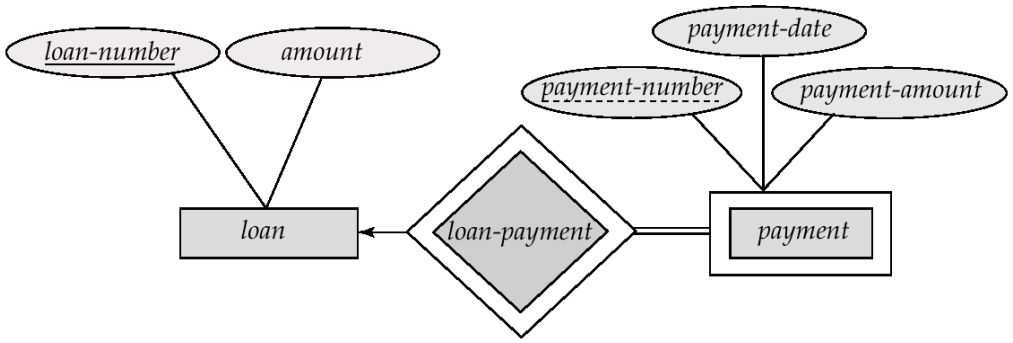
The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

We depict a weak entity set by double rectangles.

We underline the discriminator of a weak entity set with a dashed line.

payment-number – discriminator of the *payment* entity set

Primary key for *payment* – (*loan-number*, *payment-number*)



E-R diagram with a weak entity set

Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.

If *loan-number* were explicitly stored, *payment* could be made a strong entity, but then the relationship between *payment* and *loan* would be duplicated by an implicit relationship defined by the attribute *loan-number* common to *payment* and *loan*

In a university, a *course* is a strong entity and a *course-offering* can be modeled as a weak entity

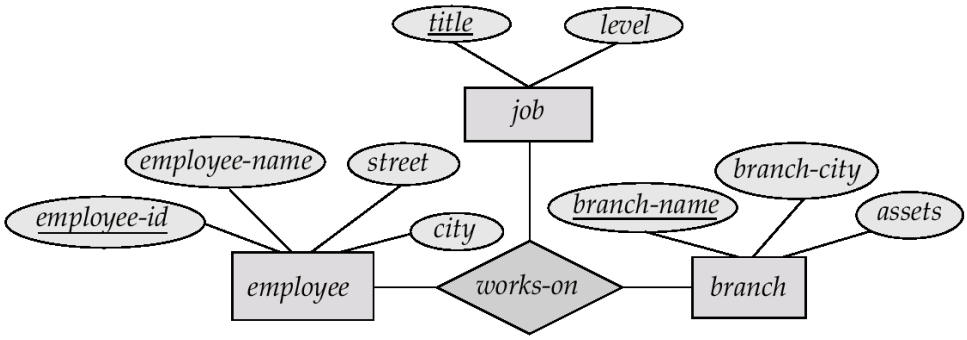
The discriminator of *course-offering* would be *semester* (including year) and *section-number* (if there is more than one section)

If we model *course-offering* as a strong entity we would model *course-number* as an attribute.

Then the relationship with *course* would be implicit in the *course-number* attribute

Nonbinary Relationships

This E-R diagram says that a customer may have several accounts, each located in a specific bank branch, and that an account may belong to several different customers.



We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint

E.g. an arrow from *works-on* to *job* indicates each employee works on at most one job at any branch.

If there is more than one arrow, there are two ways of defining the meaning.

E.g a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean

1. each *A* entity is associated with a unique entity from *B* and *C* or

2. each pair of entities from (*A*, *B*) is associated with a unique *C* entity, and each pair (*A*, *C*) is associated with a unique *B*

Each alternative has been used in different formalisms

To avoid confusion we outlaw more than one arrow

Converting Non-Binary Relationships to Binary Form

In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set. Replace R between entity sets A, B and C by an entity set E, and three relationship sets:

1. RA, relating E and A

2. RB, relating E and B

3. RC, relating E and C

Create a special identifying attribute for E

Add any attributes of R to E

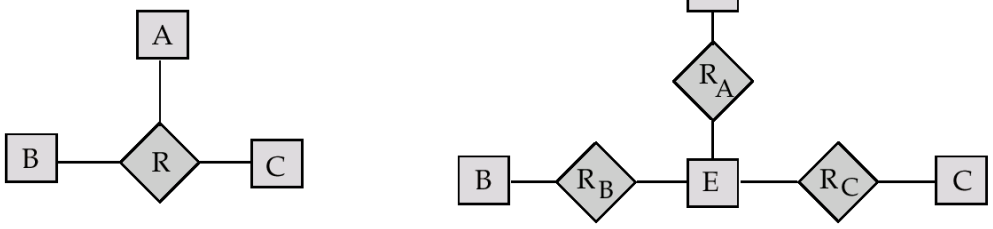
For each relationship (ai , bi , ci) in R, create

1. a new entity ei in the entity set E

2. add (ei , ai) to RA

3. add (ei , bi) to RB

4. add (ei , ci) to RC



Design of an E-R Database Scheme

The E-R data model provides a wide range of choice in designing a database scheme to accurately model some real-world situation.

Some of the decisions to be made are

- Using a ternary relationship versus two binary relationships.
- Whether an entity set or a relationship set best fit a real-world concept.
- Whether to use an attribute or an entity set.
- Use of a strong or weak entity set.
- Appropriateness of generalization.
- Appropriateness of aggregation.

Use of Extended E-R Features

We have seen weak entity sets, generalization and aggregation. Designers must decide when these features are appropriate.

- Strong entity sets and their dependent weak entity sets may be regarded as a single ``object" in the database, as weak entities are existence-dependent on a strong entity.
- It is possible to treat an aggregated entity set as a single unit without concern for its inner structure details.
- Generalization contributes to modularity by allowing common attributes of similar entity sets to be represented in one place in an E-R diagram.

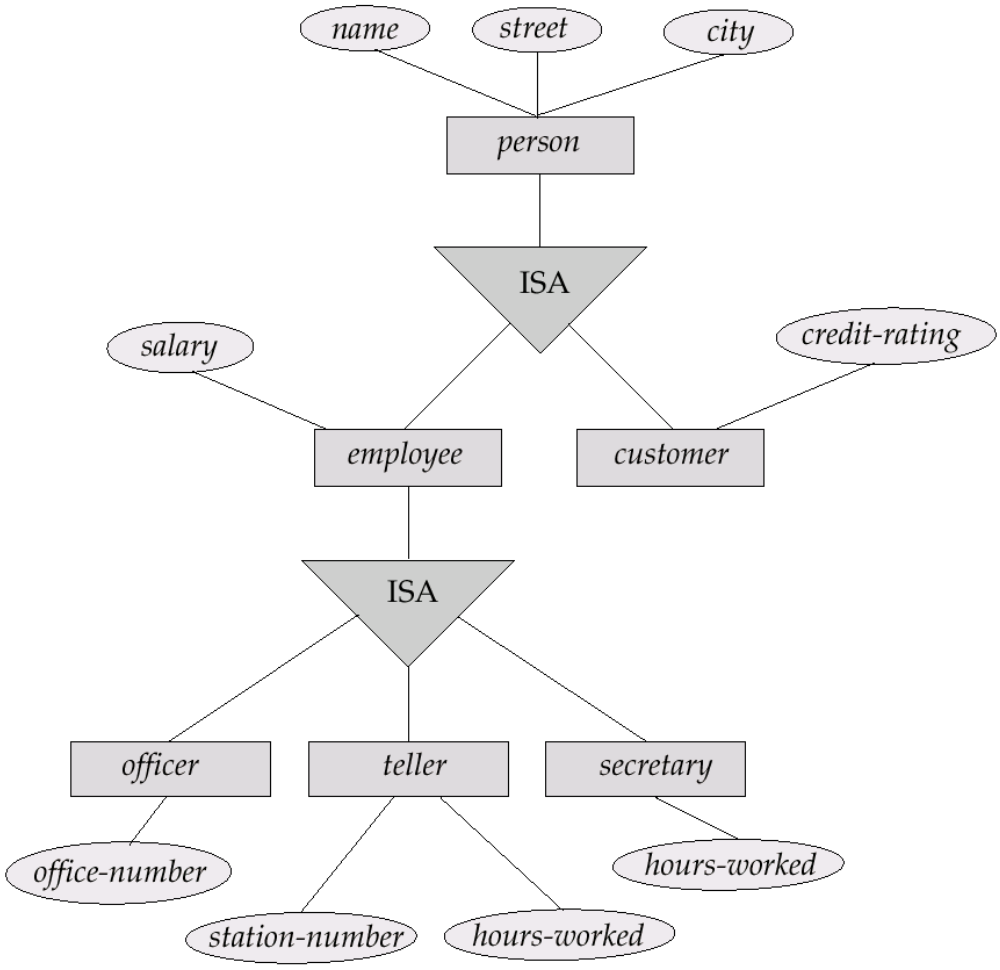
Specialization

Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.

These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.

Depicted by a *triangle* component labeled ISA (E.g. *customer* "is a" *person*).

Attribute inheritance – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.



Generalization

A bottom-up design process – combine a number of entity sets that share the same features into a higher-level entity set.

Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.

The terms specialization and generalization are used interchangeably.

Can have multiple specializations of an entity set based on different features.

E.g. *permanent-employee* vs. *temporary-employee*, in addition to *officer* vs. *secretary* vs. *teller*

Each particular employee would be

a member of one of *permanent-employee* or *temporary-employee*,

and also a member of one of *officer*, *secretary*, or *teller*

The ISA relationship also referred to as **superclass - subclass** relationship

Design Constraints on a Specialization/Generalization

Constraint on which entities can be members of a given lower-level entity set.

condition-defined

E.g. all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.

user-defined

Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.

Disjoint

an entity can belong to only one lower-level entity set

Noted in E-R diagram by writing *disjoint* next to the ISA triangle

Overlapping

an entity can belong to more than one lower-level entity set

Completeness constraint -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.

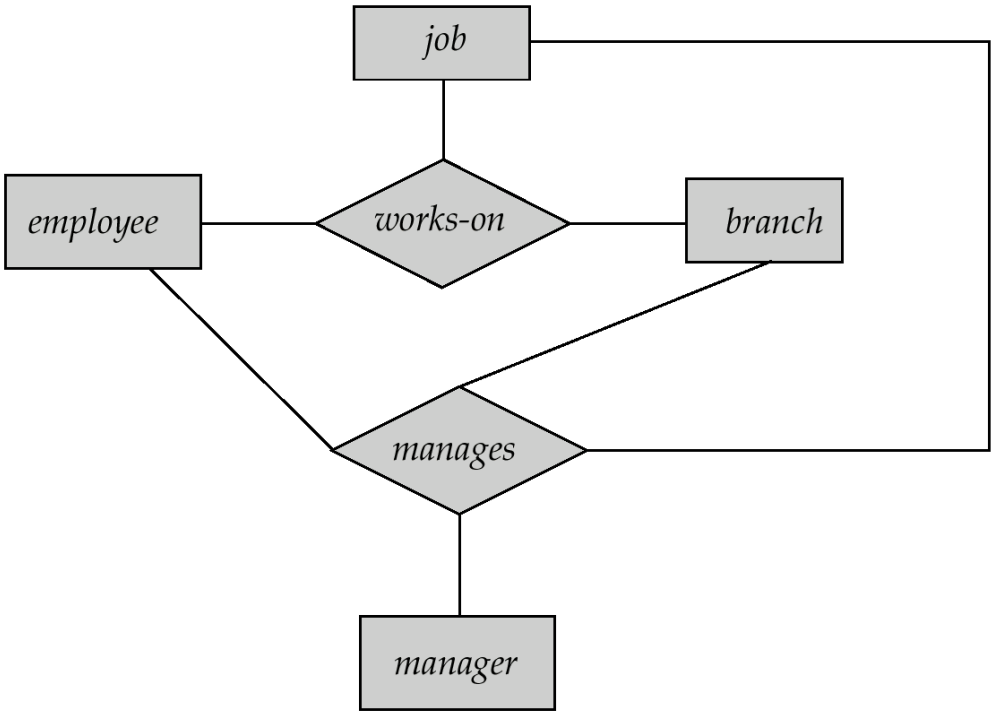
total : an entity must belong to one of the lower-level entity sets

partial: an entity need not belong to one of the lower-level entity sets

Aggregation

Consider the ternary relationship *works-on*, which we saw earlier

Suppose we want to record managers for tasks performed by an employee at a branch

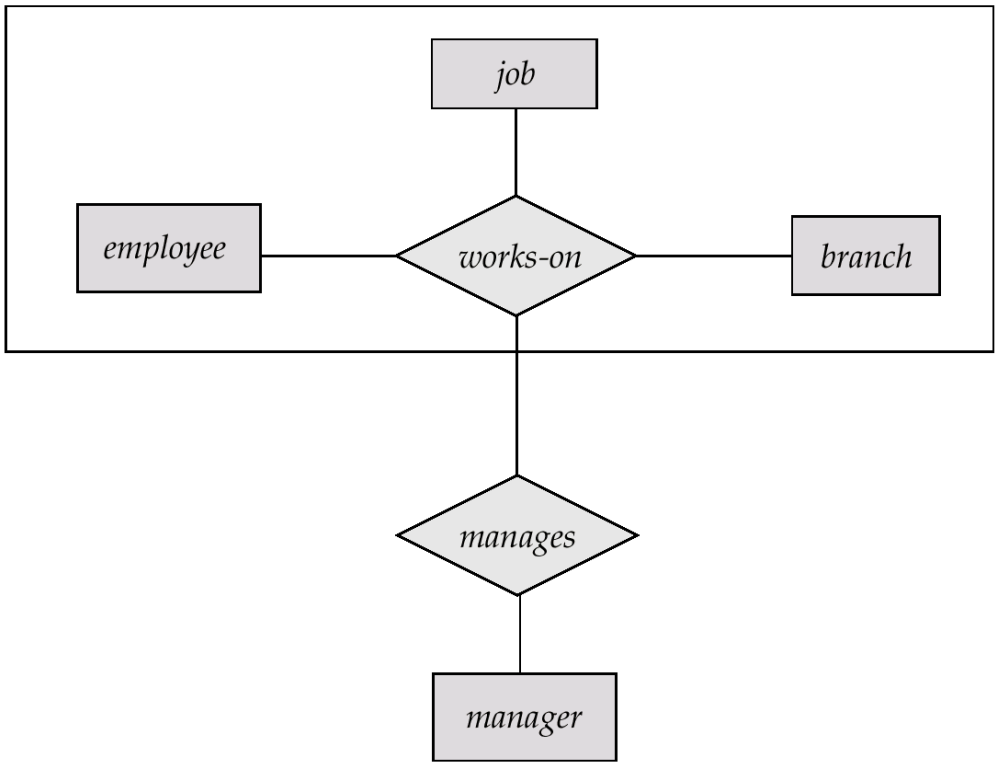


Relationship sets *works-on* and *manages* represent overlapping information
- Every *manages* relationship corresponds to a *works-on* relationship
- However, some *works-on* relationships may not correspond to any *manages* relationships
  - So we can't discard the *works-on* relationship

Eliminate this redundancy via *aggregation*
- Treat relationship as an abstract entity
- Allows relationships between relationships
- Abstraction of relationship into new entity

Without introducing redundancy, the following diagram represents:
- An employee works on a particular job at a particular branch
- An employee, branch, job combination may have an associated manager

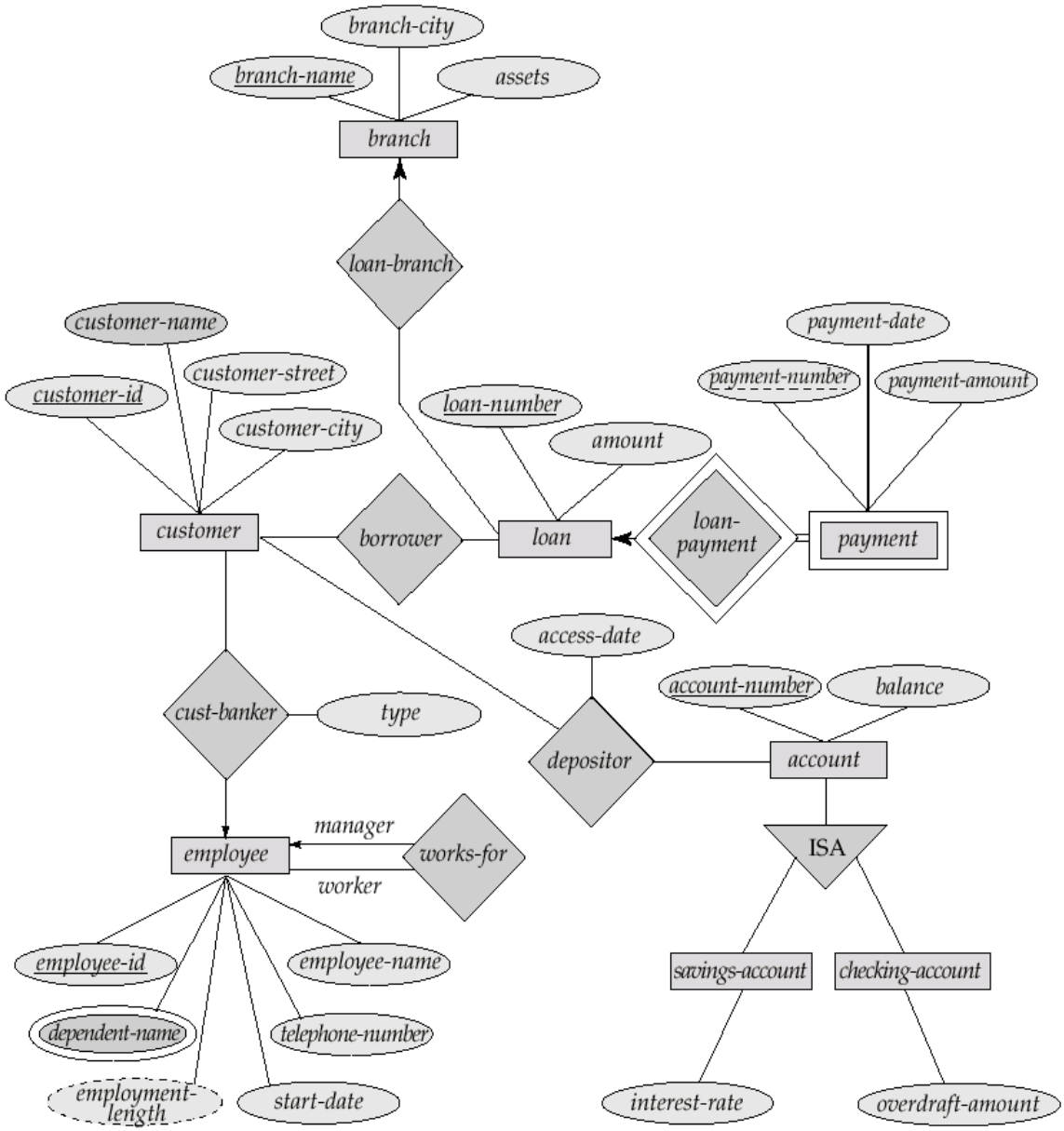
E-R Diagram With Aggregation



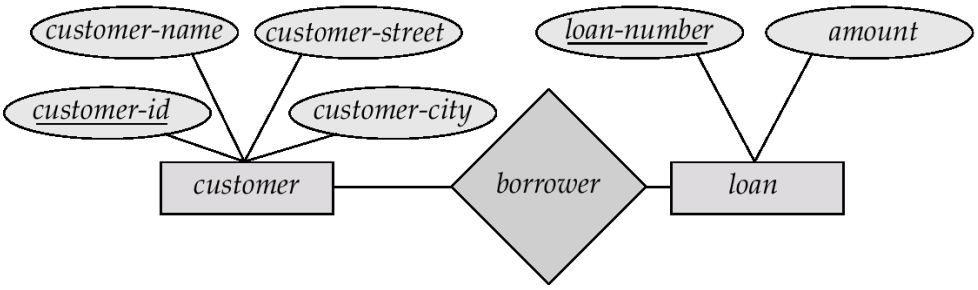
E-R Design Decisions

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



Representing Entity Sets as Tables



We use a table with one column for each attribute of the set. Each row in the table corresponds to one entity of the entity set. For the entity set *account* We can add, delete and modify rows (to reflect changes in the real world).

A row of a table will consist of an n-tuple where n is the number of attributes.

Actually, the table contains a subset of the set of all possible rows. We refer to the set of all possible rows as the **cartesian product** of the sets of all attribute values.

We may denote this as

D1 X D2

for the account table, where D1and D2 denote the set of all account numbers and all account balances, respectively.

In general, for a table of n columns, we may denote the cartesian product of D1,D2,….,Dn by

D1 X D2 X……X Dn-1 X Dn

A strong entity set reduces to a table with the same attributes

customer-id	*customer-name*	*customer-street*	*customer-city*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

Customer Table

Loan-number	amount
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Loan table

Composite and Multivalued Attributes

Composite attributes are flattened out by creating a separate attribute for each component attribute

E.g. given entity set *customer* with composite attribute *name* with component attributes *first-name* and *last-name* the table corresponding to the entity set has two attributes

name.first-name and *name.last-name*

A multivalued attribute M of an entity E is represented by a separate table EM

Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M

E.g. Multivalued attribute *dependent-names* of *employee* is represented by a table *employee-dependent-names*(*employee-id*, *dname*)

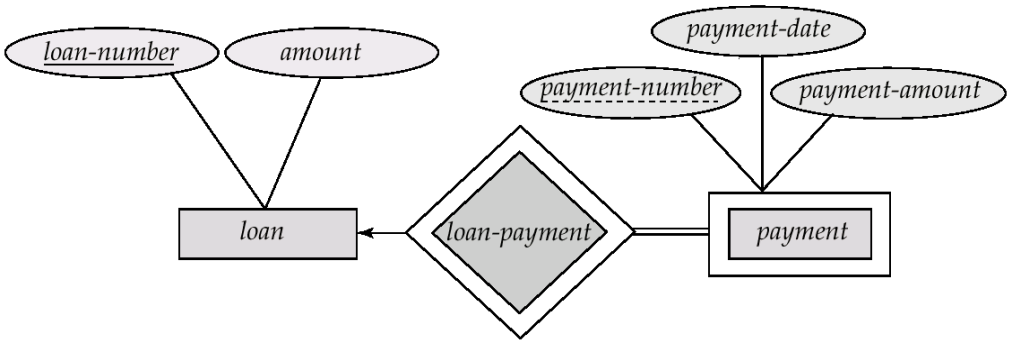
Each value of the multivalued attribute maps to a separate row of the table EM

E.g., an employee entity with primary key John and dependents Johnson and Johndotir maps to two rows:

(John, Johnson) and (John, Johndotir)

Representing Weak Entity Sets

For a weak entity set, we add columns to the table corresponding to the primary key of the strong entity set on which the weak set is dependent

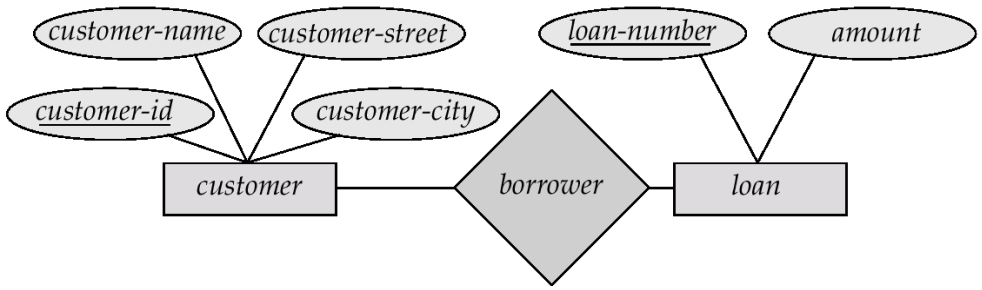


A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

loan-number	*payment-number*	*payment-date*	*payment-amount*
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

Representing Relationship Sets as Tables

A many-to-many relationship set is represented as a table with columns for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
E.g.: table for relationship set *borrower*



customer-id	*loan-number*
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

Let R be a relationship set involving entity sets $E_1, E_2, \ldots, E_m$.

The table corresponding to the relationship set R has the following attributes:

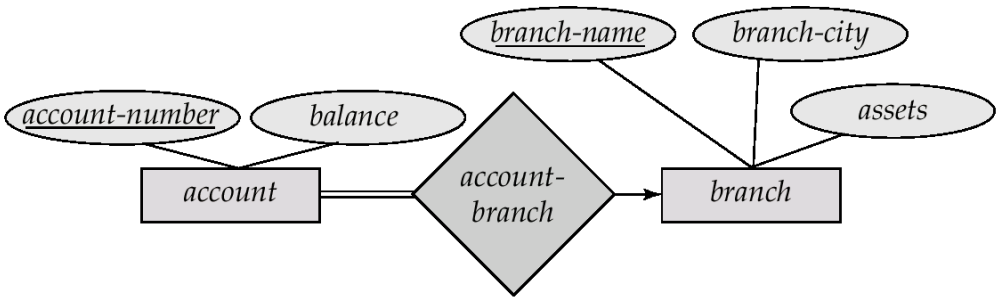
$$\bigcup_{i=1}^{m} \text{primary-key}(E_i)$$

If the relationship has k descriptive attributes, we add them too:

$$\bigcup_{i=1}^{m} \text{primary-key}(E_i) \bigcup \{a_1, a_2, \ldots, a_k\}$$

Redundancy of Tables

Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the many side, containing the primary key of the one side
E.g.: Instead of creating a table for relationship *account-branch*, add an attribute *branch* to the entity set *account*



For one-to-one relationship sets, either side can be chosen to act as the "many" side

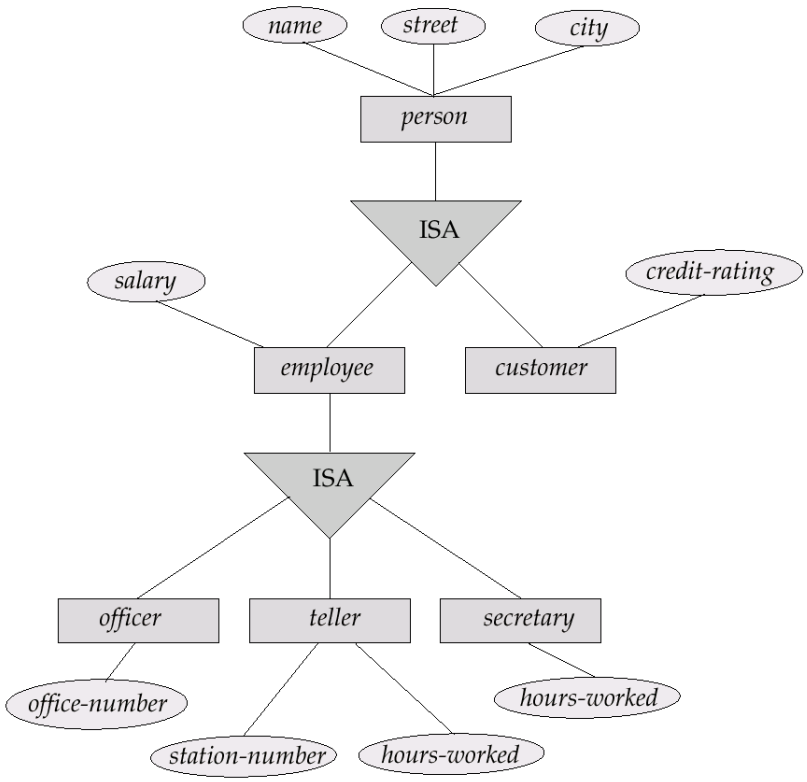
That is, extra attribute can be added to either of the tables corresponding to the two entity sets

If participation is *partial* on the many side, replacing a table by an extra attribute in the relation corresponding to the "many" side could result in null values

The table corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.

E.g. The *payment* table already contains the information that would appear in the *loan-payment* table (i.e., the columns loan-number and *payment-number*).

Representing Specialization as Tables



Method 1:

Form a table for the higher level entity

Form a table for each lower level entity set, include primary key of higher level entity set and local attributes

table	table attributes
person	*name, street, city*
customer	*name, credit-rating*
employee	*name, salary*

Drawback: getting information about, e.g., *employee* requires accessing two tables

Method 2:

Form a table for each entity set with all local and inherited attributes

table	table attributes
person	*name, street, city*
customer	*name, street, city, credit-rating*
employee	*name, street, city, salary*

If specialization is total, table for generalized entity (*person*) not required to store information

Can be defined as a "view" relation containing union of specialization tables

But explicit table may still be needed for foreign key constraints

Drawback: street and city may be stored redundantly for persons who are both customers and employees

Relations Corresponding to Aggregation

To represent aggregation, create a table containing

primary key of the aggregated relationship,

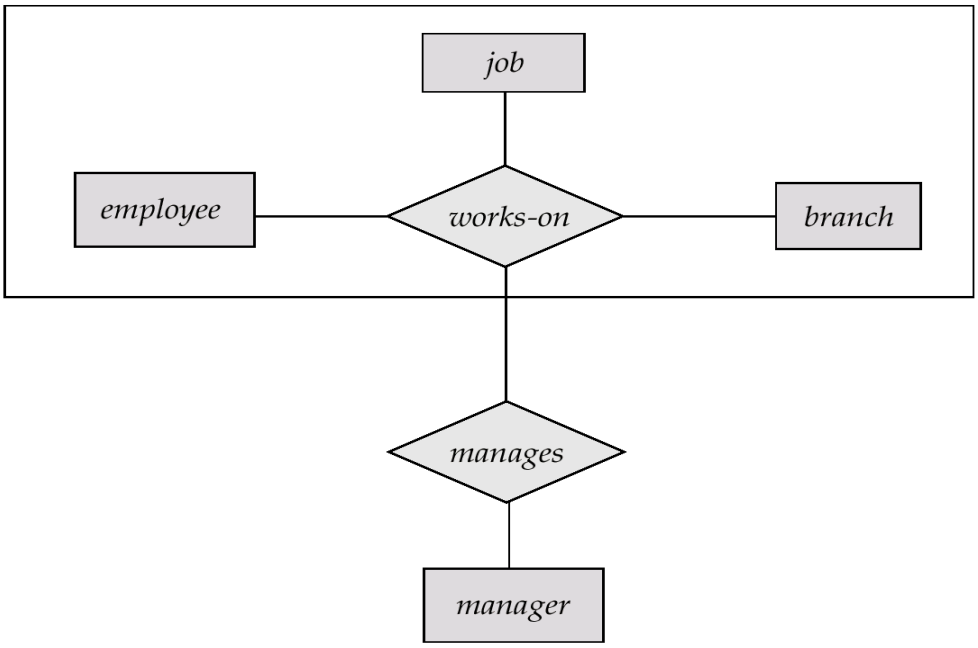
the primary key of the associated entity set

Any descriptive attributes

E.g. to represent aggregation *manages* between relationship *works-on* and entity set *manager*, create a table

manages(*employee-id, branch-name, title, manager-name*)

Table *works-on* is redundant **provided** we are willing to store null values for attribute *manager-name* in table *manages*



UNIT III

Relational Model: Structure of Relational Database, Relational Algebra, Operation, Additional Operations, Calculus, Domain Relational Calculus, Tuple Relational, Query by Examples.

Relational Algebra

Six basic operators
- select: σ
- project: ∏
- union: ∪
- set difference: –
- Cartesian product: x
- rename: ρ

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

σ A=B ^ D > 5 (r)

A	B	C	D
α	α	1	7
β	β	23	10

Select Operation

Notation: $\sigma p(r)$

p is called the **selection predicate**

Defined as:

$\sigma p(\boldsymbol{r}) = \{t \mid t \in r$ **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: =, ≠, >, ≥. <. ≤

Example of selection:

$\sigma branch\_name = \text{“Perryridge”}(account)$

Project Operation – Example

Project Operation – Example

Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

$\Pi_{A,C}$ (*r*)

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

Notation:

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

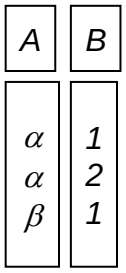
Duplicate rows removed from result, since relations are sets

Example: To eliminate the *branch_name* attribute of *account*

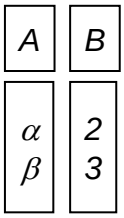
Π*account_number, balance* (*account*)

Union Operation – Example

Relations *r, s:*

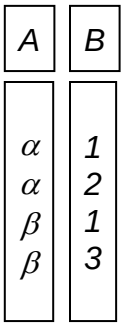


r



s

r ∪ s:



Notation: $r \cup s$

Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

For $r \cup s$ to be valid.

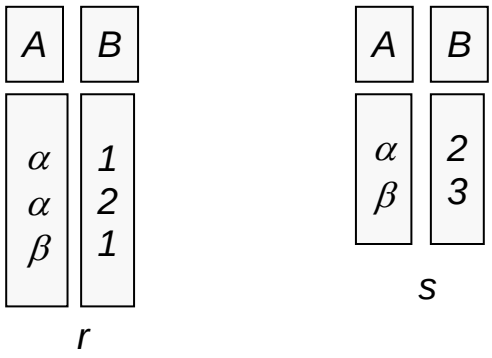
1. *r,s* must have the *same* **arity** (same number of attributes)
2. The attribute domains must be **compatible** (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)

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

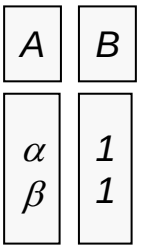
Π*customer_name* (*depositor*) $\cup \Pi$*customer_name* (*borrower)*

Set Difference Operation – Example

Relations r, s:



$r - s$:



Notation $r - s$
Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

Set differences must be taken between **compatible** relations.

r and s must have the same arity

attribute domains of r and s must be compatible

Cartesian-Product Operation – Example

Relations *r, s*:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r x *s*:

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

Cartesian-Product Operation

Notation *r* x *s*

Defined as:

$$r \times s = \{t\, q \mid t \in r \textbf{ and } q \in s\}$$

Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \varnothing$).

If attributes of *r(R)* and *s(S)* are not disjoint, then renaming must be used.

Composition of Operations

Can build expressions using multiple operations

Example: σA=C(*r x s*)

r x s

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

σA=C(*r x s*)

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

returns the result of expression *E* under the name *X*, and with the

attributes renamed to $A1 , A2 , ..., An$.

Banking Example
branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)

account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower(customer_name, loan_number)

Find all loans of over $1200

$$\sigma_{amount > 1200}(loan)$$

Find the loan number for each loan of an amount greater than $1200

$$\prod_{loan\_number}(\sigma_{amount > 1200}(loan))$$

Find the names of all customers who have a loan, an account, or both, from the bank

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

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

$$\Pi_{customer\_name}\ (\sigma_{branch\_name=\text{“}Perryridge\text{”}}\ (\sigma_{borrower.loan\_number\ =\ loan.loan\_number}(borrower\ x\ loan)))$$

Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$\Pi_{customer\_name}\ (\sigma_{branch\_name\ =\ \text{“}Perryridge\text{”}}$

$(\sigma_{borrower.loan\_number\ =\ loan.loan\_number}(\text{borrower x loan}))\ -$

$\Pi_{customer\_name}(\text{depositor})$

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

Query 1

$\Pi_{\text{customer\_name}}\ (\sigma_{\text{branch\_name = “Perryridge”}}\ (\sigma_{\text{borrower.loan\_number = loan.loan\_number}}\ (\text{borrower x loan})))$

Query 2

$\Pi_{\text{customer\_name}}(\sigma_{\text{loan.loan\_number = borrower.loan\_number}}\ ((\sigma_{\text{branch\_name = “Perryridge”}}\ (\text{loan}))\ \text{x}\ \text{borrower}))$

Find the largest account balance

Strategy:

Find those balances that are *not* the largest

- Rename *account* relation as *d* so that we can compare each account balance with all others

Use set difference to find those account balances that were *not* found in the earlier step.

The query is:

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_{d}(account)))$$

Set-Intersection Operation

Notation: $r \cap s$

Defined as:

$r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$

Assume:

r, s have the *same arity*

attributes of r and s are compatible

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

Set-Intersection Operation – Example

Relation *r, s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r∩*s*

A	B
α	2

ν Notation: r ⋈s

Let *r* and *s* be relations on schemas *R* and *S* respectively.

Then, r join s is a relation on schema $R \cup S$ obtained as follows:

Consider each pair of tuples *tr* from *r* and *ts* from *s*.
If *tr* and *ts* 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 *tr* on *r*
t has the same value as *ts* on *s*

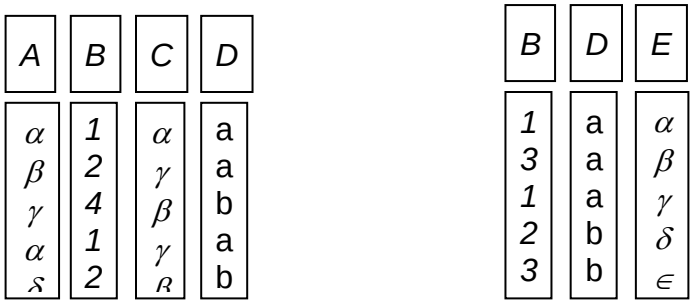
Example:
R = (*A, B, C, D*)
S = (*E, B, D*)
Result schema = (*A, B, C, D, E*)

then r join s defined as

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

Natural Join Operation – Example

Relations r, s:



R⋈ S

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Division Operation $r \div S$

Suited to queries that include the phrase "for all".
Let *r* and *s* be relations on schemas *R* and *S* respectively where

$R = (A1, \ldots, Am, B1, \ldots, Bn)$

$S = (B1, \ldots, Bn)$

The result of r ÷ s is a relation on schema

$R - S = (A1, \ldots, Am)$

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

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

Division Operation – Example

Relations *r*, *s*:

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
∈	6
∈	1
β	2

r

B
1
2

s

r÷*s*:

A
α
β

Another Division Example

Relations *r*, *s*:

A	B	C	D	E
α	a	α	a	1
α	a	γ	b	1
α	a	γ	a	1
β	a	γ	b	1
β	a	γ	a	3
γ	a	γ	b	1
γ	a	γ	a	1
γ	a	β	b	1

r

D	E
a	1
h	1

s

r÷*s*:

A	B	C
α	a	γ
γ	a	γ

Property

Let $q = r \div s$

Then q is the largest relation satisfying $q \times s \subseteq r$

Definition in terms of the basic algebra operation

Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

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

To see why

$\prod_{R-S,S}(r)$ simply reorders attributes of r

$\prod_{R-S}(\prod_{R-S}(r) \times s) - \prod_{R-S,S}(r)$ gives those tuples t in

$\prod_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.

Bank Example Queries

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

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

Find the name of all customers who have a loan at the bank and the loan amount

$$\prod_{customer\_name,\ loan\_number,\ amount}(borrower \bowtie loan)$$

Find all customers who have an account from at least the "Downtown" and the Uptown" branches.

Query 1

$\prod_{customer\_name}(\sigma_{branch\_name} = \text{"Downtown"}(depositor \text{ join } account))$

$\cap$

$\prod_{customer\_name}(\sigma_{branch\_name} = \text{"Uptown"}(depositor \text{ join } account))$

Query 2

Π*customer_name, branch_name* (*depositoraccount*)
$\div \rho$*temp(branch_name)* ({(*"Downtown"*), (*"Uptown"*)})
Note that Query 2 uses a constant relation.

Find all customers who have an account at all branches located in Brooklyn city.

Π*customer_name, branch_name* (*depositoraccount*)
$\div \Pi$*branch_name* (σ*branch_city* = "Brooklyn" (*branch*))

Extended Relational-Algebra-Operations

Generalized Projection
Aggregate Functions
Outer Join

Generalized Projection

Extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

E is any relational-algebra expression
Each of *F*1, *F*2, …, *Fn* are are arithmetic expressions involving constants and attributes in the schema of *E*.
Given relation *credit_info(customer_name, limit, credit_balance),* find how much more each person can spend:

Π*customer_name, limit – credit_balance (credit_info)*

Aggregate Functions and Operations

Aggregation function takes a collection of values and returns a single value as a result.

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate operation in relational algebra

E is any relational-algebra expression

G1, *G2* …, *Gn* is a list of attributes on which to group (can be empty)

Each *Fi* is an aggregate function

Each *Ai* is an attribute name

Relation *r*:

A	*B*	*C*
α	α	7
α	β	7
β	β	3
β	β	10

$\mathcal{G}_{\mathbf{sum(c)}}(r)$

sum(*c*)
27

Relation *account* grouped by *branch-name*:

branch name	*account number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

branch_name$\mathcal{G}$ **sum**(*balance*) (*account*)

branch name	**sum**(*balance*)
Perryridge	1300
Brighton	1500
Redwood	700

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 $\mathcal{G}$ ***sum****(balance)* **as** *sum_balance*

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.

We shall study precise meaning of comparisons with nulls later

Relation *loan*

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

Relation *borrower*

customer_na	*loan_number*
Jones	L-170
Smith	L-230
Haves	L-155

ν Join

loan ⋈ *borrower*

loan number	*branch nam*	*amount*	*customer na*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

νLeft Outer Join

loan ⟕ *borrower*

loan number	*branch nam*	*amount*	*customer na*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*

ν Right Outer Join

loan ⟖ *borrower*

loan number	*branch name*	*amount*	*customer na*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Haves

ν Full Outer Join

loan ⟗ *borrower*

loan number	*branch name*	*amount*	*customer na*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Null Values

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)
Comparisons with null values return the special truth value: *unknown*

If *false* was used instead of *unknown*, then *not (A < 5)*
would not be equivalent to *A >= 5*

Three-valued logic using the truth value *unknown*:

OR: (*unknown* **or** *true*) = *true*,
(*unknown* **or** *false*) = *unknown*
(*unknown* **or** *unknown*) = *unknown*
AND:(*true* **and** *unknown*) = *unknown*,
(*false* **and** *unknown*) = *false*,
(*unknown* **and** *unknown*) = *unknown*
NOT*:* (**not** *unknown*) = *unknown*
In SQL "*P* **is unknown**"evaluates to true if predicate *P* evaluates to *unknown*

Result of select predicate is treated as *false* if it evaluates to *unknown*

Modification of the Database

The content of the database may be modified using the following operations:

Deletion
Insertion
Updating

All these operations are expressed using the assignment operator.

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.

Deletion Examples

Delete all account records in the Perryridge branch.

$$account \leftarrow account - \sigma_{branch\_name = \text{“Perryridge”}}(account)$$

Deleteall loan records with amount in the range of 0 to 50

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$

Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{branch\_city = \text{“Needham”}}(account \bowtie branch)$$
$$r_2 \leftarrow \Pi_{account\_number, branch\_name, balance}(r_1)$$
$$r_3 \leftarrow \Pi_{customer\_name, account\_number}(r_2 \bowtie depositor)$$
$$account \leftarrow account - r_2$$
$$depositor \leftarrow depositor - r_3$$

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:

$r \leftarrow r \cup E$

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

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

Example

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

$account \leftarrow account \cup$ {(“A-973”,“Perryridge”, 1200)}

depositor $\leftarrow depositor \cup$ {(“Smith”, “A-973”)}

Example
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

$r_1 \leftarrow (\sigma_{branch\_name = \text{“Perryridge”}} (borrower \quad loan))$
$account \leftarrow account \cup \prod_{loan\_number,\ branch\_name, 200} (r_1)$
$depositor \leftarrow depositor \cup \prod_{customer\_name,\ loan\_number} (r_1)$

Updating

A mechanism to change a value in a tuple without charging *all* values in the tuple

Use the generalized projection operator to do this task

$r \leftarrow \prod_{r1,r2,\ldots\ldots rn\,(r)}$

Each F_i is either
the I^{th} attribute of r, if the I^{th} attribute is not updated, or,
if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute

Example

Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \prod_{account\_number,\ branch\_name,\ balance * 1.05} (account)$$

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

$$account \leftarrow \prod_{account\_number,\ branch\_name,\ balance * 1.06} (\sigma_{BAL > 10000} (account)) \cup \prod_{account\_number,\ branch\_name,\ balance * 1.05} (\sigma_{BAL \leq 10000} (account))$$

Views

In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount.

$\Pi_{name,loan\text{-}number,bramch\text{-}name}$ (borrower X loan)

A **view** provides a mechanism to hide certain data from the view of certain users.
Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

Create view v as < query expression>

Where < query expression> is any legal relational algebra query expression. The view name is represented by v

Example

Consider the view (named *all-customer*) consisting of branches and their customers.

create view *all-customer* as
Π *branch-name, customer-name(depositor X account)*
U Π*branch-name, customer-name*(*borrower X loan*)

We can find all customers of the Perryridge branch by

Π***customer-name***(**σ*branch-name*= "Perryridge"(*all-customer*))**

Updates Through View

• Database modifications expressed as views must be translated to modifications of the actual relations in the database.

• Consider the person who needs to see all loan data in the *loan*relation except *amount.*The view given to the person, *branch-loan,* is defined as:

create view *branch-loan* **as**
Π*branch-name, loan-number(loan)*

•Since we allow a view name to appear wherever a relation name is allowed, the person may write:

branch-loan ←*branch-loan* U{("Perryridge", L-37)}

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation *v*1is said to *depend directly*on a view relation *v2*if *v*2is used in the expression defining *v*1
- A view relation *v*1is said to *depend on*view relation *v2*if either v1 depends directly to *v*2 or there is a path of dependencies from v1to v2
- A view relation *v*is said to be *recursive*if it depends on itself.

Why do we use Relational Algebra?

Because:
•It is mathematically defined (where relations are sets)
•We can prove that two relational algebra expressions are equivalent. For example:

σcond1(σcond2R)≡σcond2(σcond1R)≡σcond1 and cond2R

R1 ⋈condR2≡σcond(R1 X R2)
R1 ÷R2≡πx(R1) -πx((πxR1) X R2) -R1)

Uses of Relational Algebra Equivalences

•To help query writers -they can write queries in several different ways
• To help query optimizers -they can choose among different ways to execute the query
and in both cases we know for surethat the two queries (the original and the replacement) are identical…that they will produce the same answer

Tuple Relational Calculus

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
t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
$t \in r$ denotes that tuple t is in relation r
P is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<, \leq, =, \neq, >, \geq$)
3. Set of connectives: and ($\wedge$), or (v), not ($\neg$)
4. Implication ($\Rightarrow$): x $\Rightarrow$ y, if x if true, then y is true

   $x \Rightarrow y \equiv \neg x \vee y$
5. 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

Banking Example

branch (*branch_name, branch_city, assets*)
customer (*customer_name, customer_street, customer_city*)
account (*account_number, branch_name, balance*)
loan (*loan_number, branch_name, amount*)
depositor (*customer_name, account_number*)
borrower(*customer_name, loan_number*)

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

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

Find the loan number for each loan of an amount greater than $1200

$$\{t \mid \exists s \in \text{loan}\ (t[loan\_number] = s[loan\_number] \wedge s[amount] > 1200)\}$$

Notice that a relation on schema [*loan_number*] is implicitly defined by the query

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 who have a loan and an account at the bank

$\{t \mid \exists s \in borrower\ (\ t\ [customer\_name\ ] = s\ [customer\_name\ ])$
$\wedge \exists u \in depositor\ (\ t\ [customer\_name\ ] = u\ [customer\_name] \ )$

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 who have a loan at the Perryridge branch, but no account at any branch of the bank

$\{t \mid \exists s \in borrower\ (t\ [customer_name\ ] = s\ [$customer_name]
$\wedge \exists u \in loan\ (u\ [branch\_name\ ] =$ "Perryridge"
$\wedge u\ [loan_number\ ] = s\ [$loan_number]))
$\wedge$ **not** $\exists v \in depositor\ (v\ [customer\_name\ ] =$
$t\ [customer\_name\ ])\}$

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

Domain Relational Calculus

A nonprocedural query language equivalent in power to the tuple relational calculus
Each query is an expression of the form:

$$\{ \langle x1, x2, \ldots, xn \rangle \mid P(x1, x2, \ldots, xn)\}$$

$x1, x2, \ldots, xn$ represent domain variables
P represents a formula similar to that of the predicate calculus

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 Perryridge branch and the loan amount:

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

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

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

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

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

Safety of Expressions

The expression:

$$\{ \langle x1, x2, \ldots, xn \rangle \mid P(x1, x2, \ldots, xn)\}$$

is safe if all of the following hold:

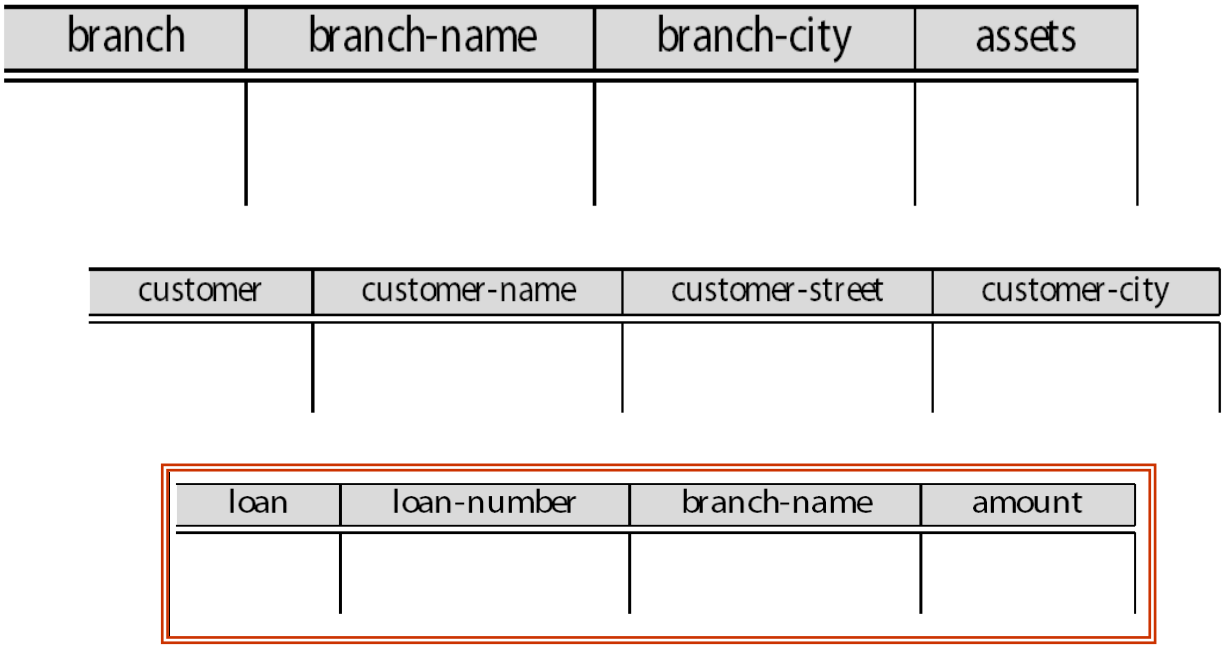
1. All values that appear in tuples of the expression are values from $dom(P)$ (that is, the values appear either in P or in a tuple of a relation mentioned in P).
2. For every “there exists” subformula of the form $\exists x\ (P1(x))$, the subformula is true if and only if there is a value of x in $dom(P1)$ such that $P1(x)$ is true.
3. For every “for all” subformula of the form $\forall x\ (P1\ (x))$, the subformula is true if and only if $P1(x)$ is true for all values x from $dom(P1)$.

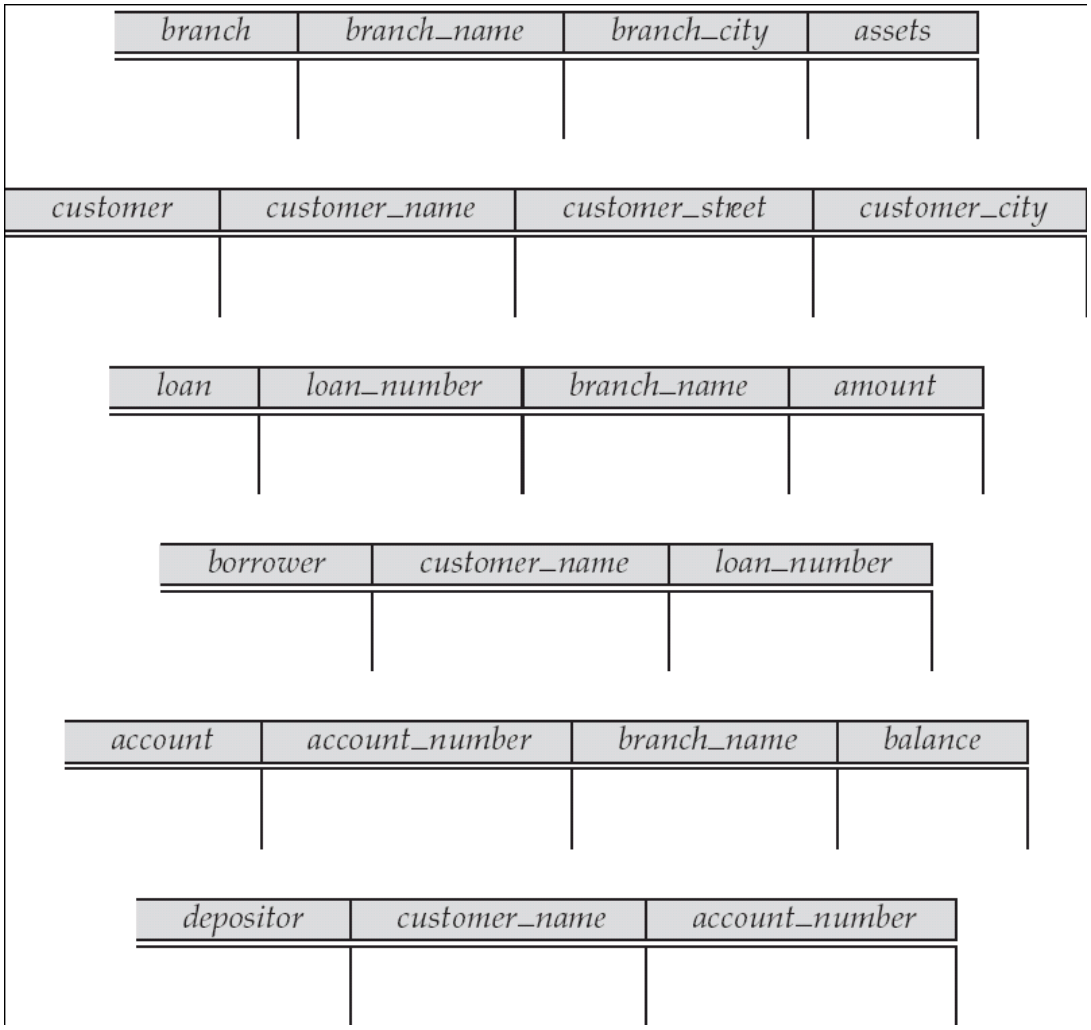
Query-by-Example (QBE)

QBE — Basic Structure

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




branch
branch_name
branch_city
assets
customer
customer_name
customer_street
customer_city
loan
loan_number
branch_name
amount
borrower
customer_name
loan_number
account
account_number
branch_name
balance
depositor
customer_name
account_number

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	

Display full details of all loans

Method 1:

loan	*loan-number*	*branch-name*	*amount*
	P._x	P._y	P._z

Method 2: Shorthand notation

loan	*loan_number*	*branch_name*	*amount*
P.			

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	P._*x*
	Jones	_*x*

Find all customers who live in the same city as Jones

customer	*customer_name*	*customer_street*	*customer_city*
	P._*x*		_*y*
	Jones		_*y*

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.

depositor	*customer_name*	*account_number*
	P._x	

borrower	*customer_name*	*loan_number*
¬	_x	

¬ means “there does not exist”

Find all customers who have at least two accounts.

depositor	*customer_name*	*account_number*
	P._x	_y
	_x	¬ _y

¬ means “not equal to”

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 the loan numbers of all loans made to Smith, to Jones, or to both jointly

borrower	*customer_name*	*loan_number*
	_*n*	P._*x*

conditions
_*n* = Smith **or** _*n* = Jones

QBE supports an interesting syntax for expressing alternative values

branch	*branch_name*	*branch_city*	*assets*
	P.	_x	

conditions
_x = (Brooklyn **or** Queens)

Find all account numbers with a balance greater than $1,300 and less than $1,500

account	*account_number*	*branch_name*	*balance*
	P.		_x

conditions
_x ≥ 1300
_x ≤ 1500

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

account	*account_number*	*branch_name*	*balance*
	P.		_x

conditions
_x = (≥ 1300 **and** ≤ 2000 **and** ¬ 1500)

Find all branches that have assets greater than those of at least one branch located in Brooklyn

branch	*branch_name*	*branch_city*	*assets*
	P._x	Brooklyn	_y _z

conditions
_y > _z

Find the *customer_name*, *account_number*, and *balance* for all customers who have an account at the Perryridge branch.

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).

Aggregate Operations

The aggregate operators are AVG, MAX, MIN, SUM, and CNT

The above operators must be postfixed with “ALL” (e.g., SUM.ALL. or AVG.ALL._x) to ensure that duplicates are not eliminated.
Example: Find the total balance of all the accounts maintained at the Perryridge branch.

account	*account_number*	*branch_name*	*balance*
		Perryridge	P.SUM.ALL.

UNQ is used to specify that we want to eliminate duplicates
Find the total number of customers having an account at the bank.

depositor	*customer_name*	*account_number*
	P.CNT.UNQ.	

Query Examples

Find the average balance at each branch.

account	*account_number*	*branch_name*	*balance*
		P.G.	P.AVG.ALL._x

The “G” in “P.G” is analogous to SQL’s **group by** construct

The “ALL” in the “P.AVG.ALL” entry in the *balance* column ensures that all balances are considered

To find the average account balance at only those branches where the average account balance is more than $1,200, we simply add the condition box:

conditions
AVG.ALL._$x > 1200$

Find all customers who have an account at all branches located in Brooklyn

depositor	customer_name	account_number
	P.G._x	_y

account	account_number	branch_name	balance
	_y	_z	

branch	branch_name	branch_city	assets
	_z	Brooklyn	
	_w	Brooklyn	

conditions
CNT.UNQ._z = CNT.UNQ._w

Modification of the Database – Deletion

Deletion of tuples from a relation is expressed by use of a D. command. In the case where we delete information in only some of the columns, null values, specified by –, are inserted.

Delete customer Smith

customer	customer_name	customer_street	customer_city
D.	Smith		

Delete the *branch_city* value of the branch whose name is “Perryridge”.

branch	branch_name	branch_city	assets
	Perryridge	D.	

Delete all loans with a loan amount greater than $1300 and less than $1500.
For consistency, we have to delete information from loan and borrower tables

loan	*loan_number*	*branch_name*	*amount*
D.	_y		_x

borrower	*customer_name*	*loan_number*
D.		_y

conditions
$\_x = (\geq 1300$ **and** $\leq 1500)$

Delete all accounts at branches located in Brooklyn.

account	*account_number*	*branch_name*	*balance*
D.	_y	_x	

depositor	*customer_name*	*account_number*
D.		_y

branch	*branch_name*	*branch_city*	*assets*
	_x	Brooklyn	

Modification of the Database – Insertion

Insertion is done by placing the I. operator in the query expression.
Insert the fact that account A-9732 at the Perryridge branch has a balance of $700.

account	*account_number*	*branch_name*	*balance*
I.	A-9732	Perryridge	700

Modification of the Database – Insertion

Provide as a gift for all loan customers of the Perryridge branch, a new $200 savings account for every loan account they have, with the loan number serving as the account number for the new savings account.

account	*account_number*	*branch_name*	*balance*
I.	_x	Perryridge	200

depositor	*customer_name*	*account_number*
I.	_y	_x

loan	*loan_number*	*branch_name*	*amount*
	_x	Perryridge	

borrower	*customer_name*	*loan_number*
	_y	_x

Modification of the Database – Updates

Use the U. operator to change a value in a tuple without changing *all* values in the tuple. QBE does not allow users to update the primary key fields.

Update the asset value of the Perryridge branch to $10,000,000.

branch	*branch_name*	*branch_city*	*assets*
	Perryridge		U.10000000

ν Increase all balances by 5 percent.

account	*account_number*	*branch_name*	*balance*
			U._*x* * 1.05