What is a Database Management System?

- A *Database Management System (DBMS)* is a software package designed to store and manage databases:
  1. Manages very large amounts of data.
  2. Supports efficient access to very large amounts of data.
  3. Supports concurrent access to very large amounts of data.
     - Example: bank and its ATM machines.
  4. Supports secure, atomic access to very large amounts of data.
     - Contrast two people editing the same UNIX file – last to write “wins” – with the problem if two people deduct money from the same account via ATM machines at the same time – new balance is wrong whichever writes last.

Example: Online Bookseller

- Data = information on books (including categories, bestsellers, etc.), customers, pending orders, order histories, trends and preferences, etc.
  - Massive: many gigabytes at a minimum for medium-size bookseller, more if keep all order histories over all time, even more if keep images of book covers and sample pages

    => Far too big for memory
  - Persistent: data outlives programs that operate on it
  - Multi-user: many people/programs accessing same database, or even same data, simultaneously

    => Need careful controls

Files vs. DBMS

- Application must stage large datasets between main memory and secondary storage (e.g., buffering, page-oriented access, 32-bit addressing, etc.)
- Special code for different queries
- Must protect data from inconsistency due to multiple concurrent users
- Crash recovery
- Security and access control

What is a Relational Database?

- Based on the relational model (tables):

acct #	name	balance
12345	Sally	1000.21
34567	Sue	285.48
…	…	…

- Today used in *most* DBMS's.

The DBMS Marketplace

- Relational DBMS companies – Oracle, Sybase – are among the largest software companies in the world.
- IBM offers its relational DB2 system. With IMS, a nonrelational system, IBM is by some accounts the largest DBMS vendor in the world.
- Microsoft offers SQL-Server, plus Microsoft Access for the cheap DBMS on the desktop, answered by "lite" systems from other competitors.
- Relational companies also challenged by "object-oriented DB" companies.
- But countered with "object-relational" systems, which retain the relational core while allowing type extension as in OO systems.

Three Aspects to Studying DBMS's

1. Modeling and design of databases.
   – Allows exploration of issues before committing to an implementation.
2. Programming: queries and DB operations like update.
   – SQL = "intergalactic dataspeak."
3. DBMS implementation.

Query Languages

Employee	
Name	Dept

Department	
Dept	Manager

SQL

SELECT Manager
FROM Employee, Department
WHERE Employee.name = "Clark Kent”
AND Employee.Dept = Department.Dept

Query Language

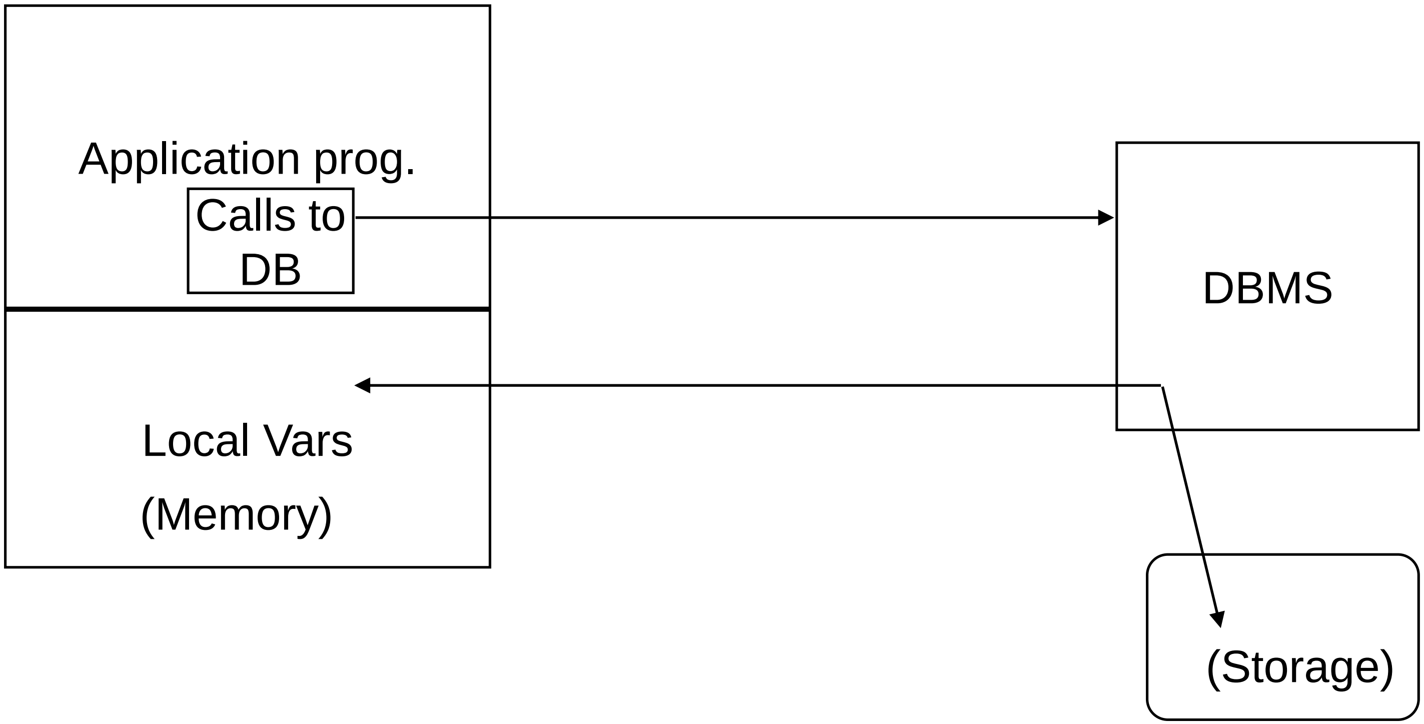
Data definition language (DDL) ~ like type defs in C or Pascal

Data Manipulation Language (DML)

Query (SELECT)
UPDATE < relation name >
SET <attribute> = < new-value>
WHERE <condition>

Host Languages

C, C++, Java, Lisp, COBOL



- Host language is completely general (Turing complete)
- Query language—less general "non procedural" and optimizable

Relational Model

Relational model is good for:

- Large amounts of data —> simple operations
- Navigate among small number of relations

Difficult Applications for relational model:

- VLSI Design (CAD in general)
- CASE
- Graphical Data

Other Models

Where number of "relations" is large, relationships are complex

- Object Data Model
- Logic Data Model

OBJECT DATA MODEL

1. Complex Objects – Nested Structure (pointers or references)
2. Encapsulation, set of Methods/Access functions
3. Object Identity
4. Inheritance – Defining new classes like old classes

Object model: usually find objects via explicit navigation

Also query language in some systems

Other Models

LOGIC (Horn Clause) DATA MODEL

- Prolog, Datalog:

  if A1 and A2 then B

  B:- A1 and A2

- Functions s(5) = 6 (successor)
- Predicates with Arguments: sum(X,Y,Z) $\leftarrow$ X + Y = Z

  sum(X,0,X) means X + 0 = X (always true for all X)

  sum(X,s(Y),s(Z)):-sum(X,Y,Z) means X+(Y+1) = (Z+1) if X + Y = Z
- More powerful than relational

  Can Compute Transitive Closure

  edge(X,Y).

  path(X,Y) :- edge(X,Y).

  path(X,Z) :- path(X,Y) & edge(Y,Z).

Data Models

60's

Hierarchical | Network

70's

80's

Relational

Choice for most new applications

90's

Object Bases | Knowledge Bases

now

Why Use a DBMS?

- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration.
- Concurrent access, recovery from crashes.

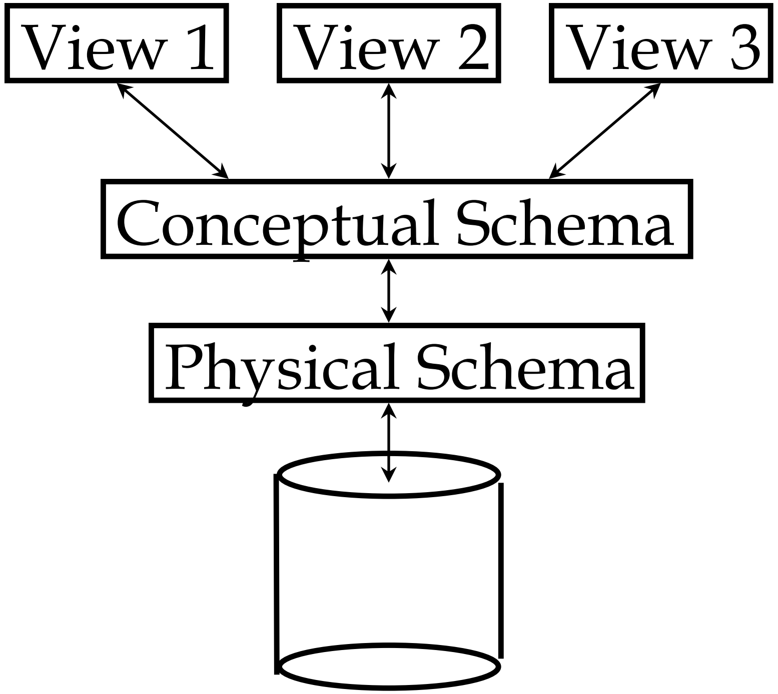
Data Independence *

- Applications insulated from how data is structured and stored.
- *Logical data independence*: Protection from changes in *logical* structure of data.
- *Physical data independence*: Protection from changes in *physical* structure of data.

☒ *One of the most important benefits of using a DBMS!*

Levels of Abstraction

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.



☒ *Schemas are defined using DDL; data is modified/queried using DML.*

Concurrency Control

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- Interleaving actions of different user programs can lead to inconsistency: e.g., cheque is cleared while account balance is being computed.
- DBMS ensures such problems don’t arise: users can pretend they are using a single-user system.

Transaction: An Execution of a DB Program

- Key concept is *transaction*, which is an *atomic* sequence of database actions (reads/writes).
- Each transaction, executed completely, must leave the DB in a *consistent state* if DB is consistent when the transaction begins.
  - Users can specify some simple *integrity constraints* on the data, and the DBMS will enforce these constraints.
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
  - Thus, ensuring that a transaction (run alone) preserves consistency is ultimately the user's responsibility!

Scheduling Concurrent Transactions

- DBMS ensures that execution of {T1, ... , Tn} is equivalent to some *serial* execution T1' ... Tn'.
  - Before reading/writing an object, a transaction requests a lock on the object, and waits till the DBMS gives it the lock. All locks are released at the end of the transaction. (Strict 2PL locking protocol.)
  - Idea: If an action of Ti (say, writing X) affects Tj (which perhaps reads X), one of them, say Ti, will obtain the lock on X first and Tj is forced to wait until Ti completes; this effectively orders the transactions.
  - What if Tj already has a lock on Y and Ti later requests a lock on Y? (Deadlock!) Ti or Tj is aborted and restarted!

Ensuring Atomicity

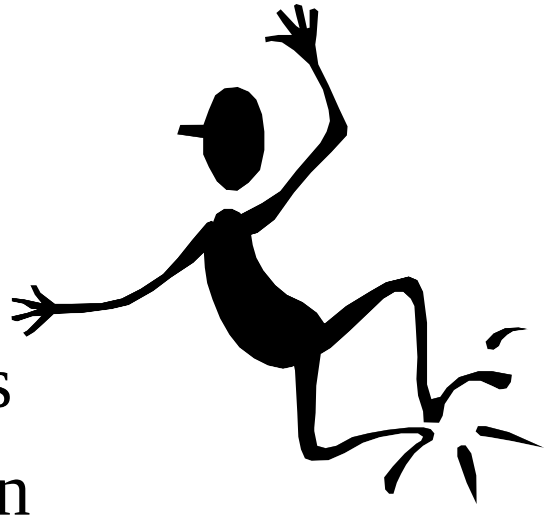
- DBMS ensures *atomicity* (all-or-nothing property) even if system crashes in the middle of a Xact.
- Idea: Keep a *log* (history) of all actions carried out by the DBMS while executing a set of Xacts:
  - Before a change is made to the database, the corresponding log entry is forced to a safe location. (*WAL protocol*; OS support for this is often inadequate.)
  - After a crash, the effects of partially executed transactions are *undone* using the log. (Thanks to WAL, if log entry wasn't saved before the crash, corresponding change was not applied to database!)

The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: The old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: A log record indicating this action.
- Log records chained together by Xact id, so it's easy to undo a specific Xact (e.g., to resolve a deadlock).
- Log is often *duplexed* and *archived* on "stable" storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Databases make these folks happy

- End users and DBMS vendors
- DB application programmers
  - e.g., smart webmasters
- *<u>Database administrator (DBA)</u>*
  - Designs logical /physical schemas
  - Handles security and authorization
  - Data availability, crash recovery
  - Database tuning as needs evolve

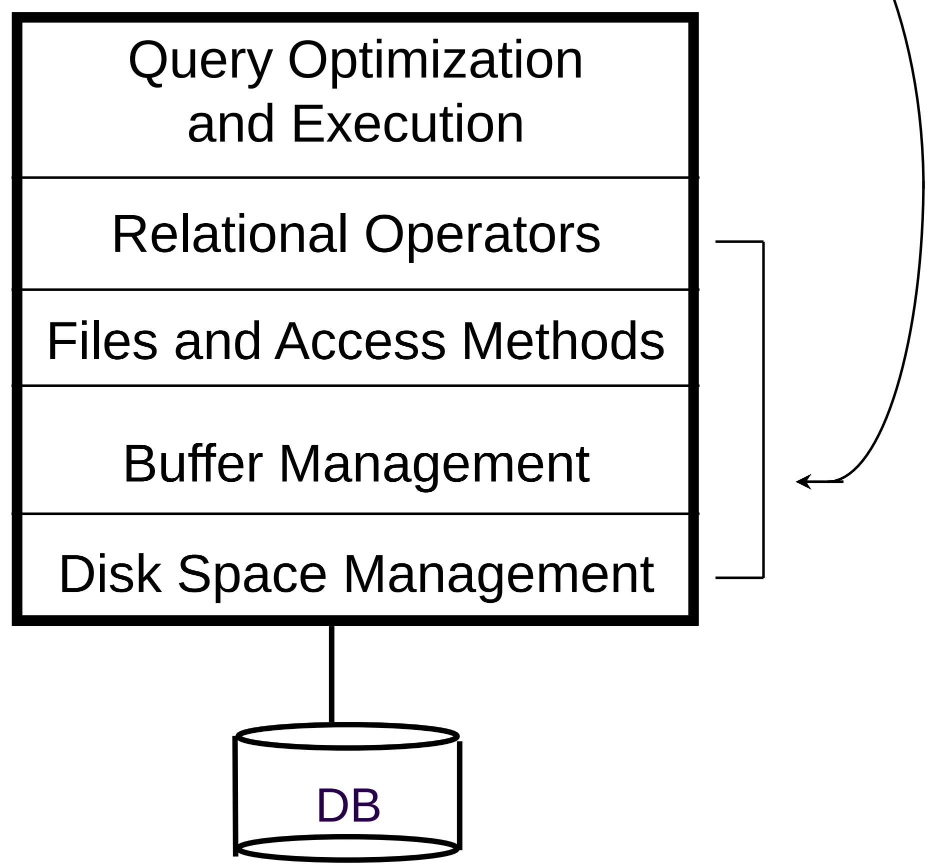


Must understand how a DBMS works!

Structure of a DBMS

These layers must consider concurrency control and recovery

- A typical DBMS has a layered architecture.
- The figure does not show the concurrency control and recovery components.
- This is one of several possible architectures; each system has its own variations.



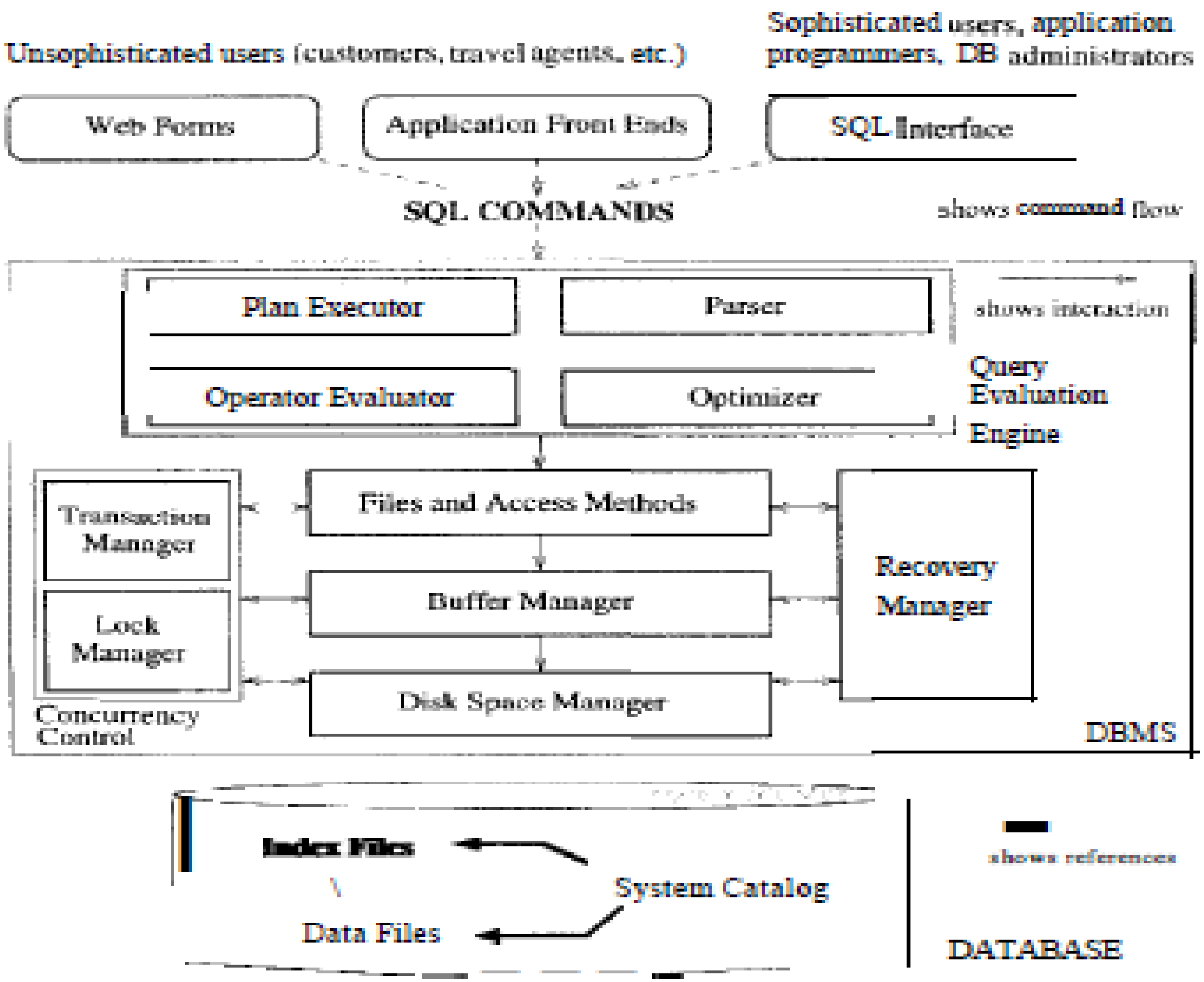
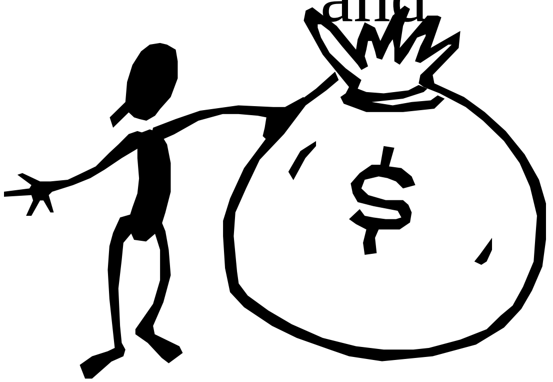


Figure 1.3 Architecture of a DBMS

Summary

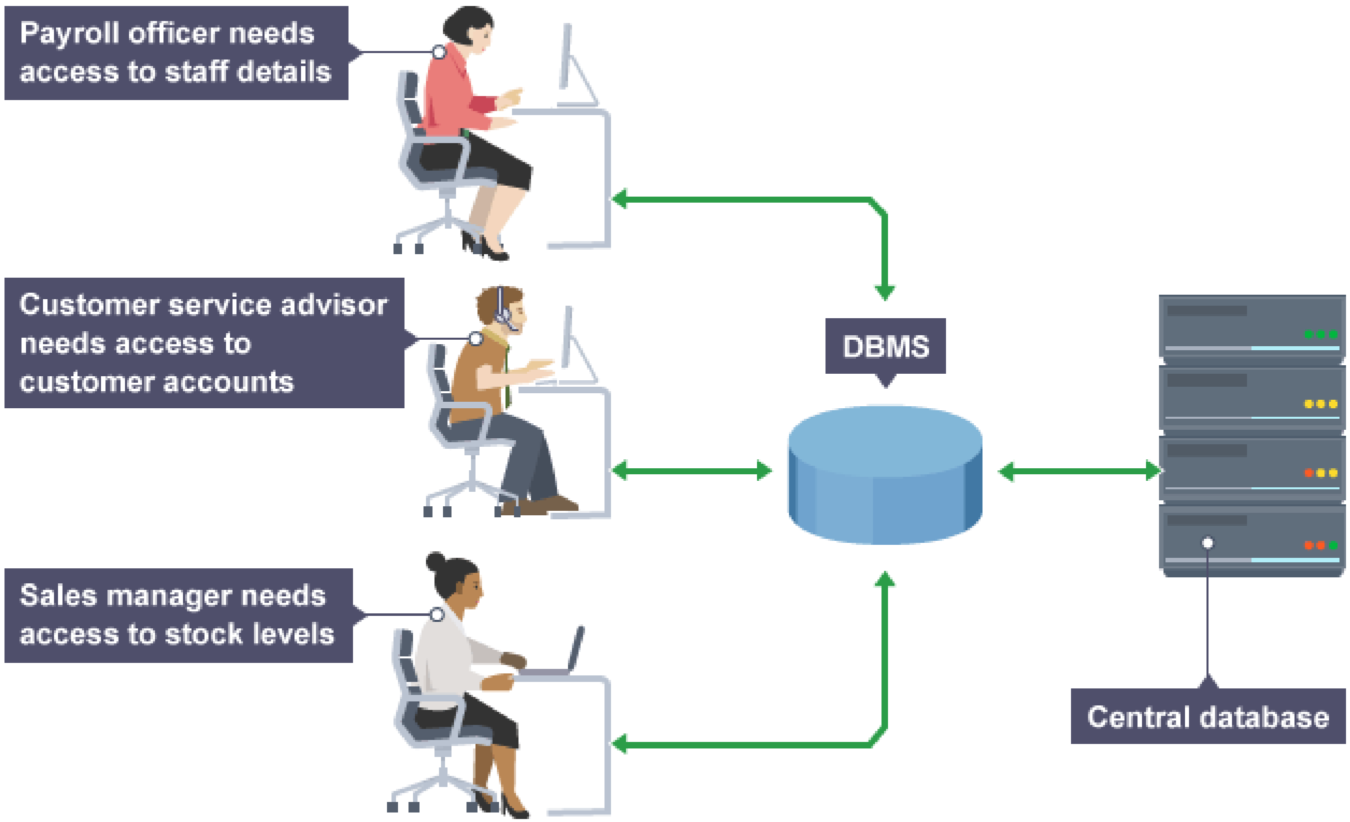
- DBMS used to maintain, query large datasets.
- Benefits include recovery from system crashes, concurrent access, quick application development, data integrity and security.
- Levels of abstraction give data independence.
- A DBMS typically has a layered architecture.
- DBAs hold responsible jobs and are well-paid! ☺
- DBMS R&D is one of the broadest, most exciting areas in CS.



The Entity-Relationship Model

Data Model

- Move from informal description of what user wants to
- Precise description of what can be implemented in a DBMS



Payroll officer needs access to staff details
Customer service advisor needs access to customer accounts
Sales manager needs access to stock levels
DBMS
Central database

Steps in developing a database

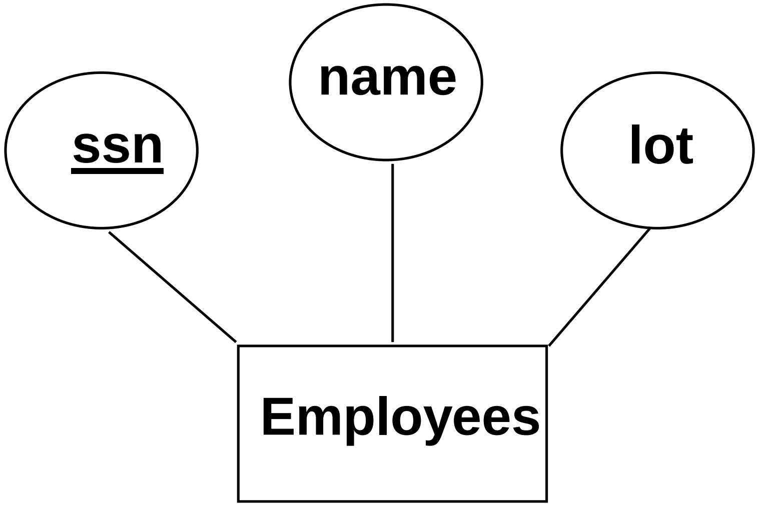
- Requirements analysis →
- Conceptual Database design →
- Logical Database design →
- Schema refinement →
- Physical Database design
- Applications and security

<u>Conceptual design</u>: (ER Model is used at this stage.)

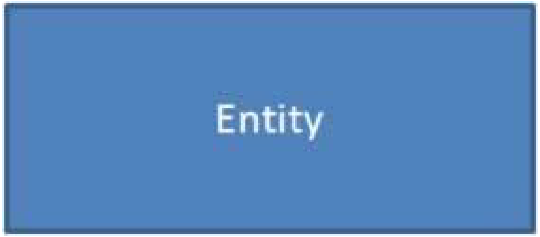
- What are the *entities* and *relationships* in the enterprise?
- What information about these entities and relationships should we store in the database?
- What are the *integrity constraints* or *business rules* that hold?
- A database `schema' in the ER Model can be represented pictorially (*ER diagrams*).
- Can map an ER diagram into a relational schema.

- The ER diagram is just an approximate description of the data, constructed through a subjective evaluation of the information collected during requirements analysis.
- A more careful analysis can often refine the logical schema obtained at the end of Step 3

ER Model Basics



- *Entity:* Real-world object distinguishable from other objects. An entity is described using a set of *attributes*. Each attribute has a *domain*.
- *Entity Set*: A collection of similar entities. E.g., all employees.
  - All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)
  - Each entity set has a *key*.



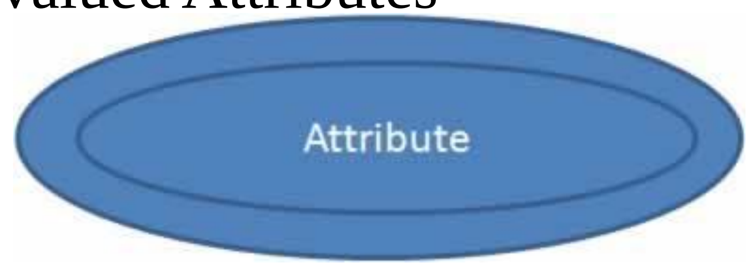
- Two classes of entities are there:
  - Regular Entity or Strong Entity
  - Weak Entity

Attribute

Properties/characteristics which describe entities are called attributes.

Key Attribute

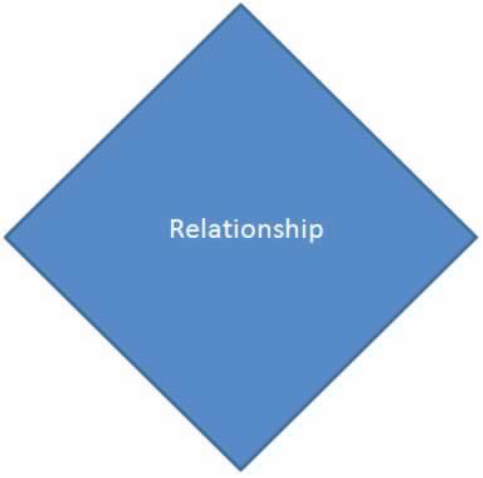
Multivalued Attributes



Derived Attribute

Relationships

Associations between entities are called relationships



Keys

- Minimal set of attributes which uniquely identify an instance of a entity

- Many candidate keys choose one to be a primary keys

- SSN vs Name … key must be unique

Keys

Types of keys in DBMS

Primary Key – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table. (ex: In a student table SRN or studID)

Super Key – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table. (ex: **Employee table (Emp_SSN, Emp_num, Emp_name))**

Candidate Key – A super key with no redundant attribute is known as candidate key, Candidate key is also called as minimal super key. Primary keys can be selected from Candidate keys.

Alternate Key – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

Composite Key – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.

Foreign Key – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

for better understanding of the key. Table “**Branch Info**”

	Branch_Id	Branch_Name	Branch_Code
	1	Computer Scie...	CSE
	2	Electronics	ECE
	3	Mechnical	MCE
	4	Information Te...	ITE
	5	Civil	CVE
▸*	*NULL*	*NULL*	*NULL*

Table: Branch_Info

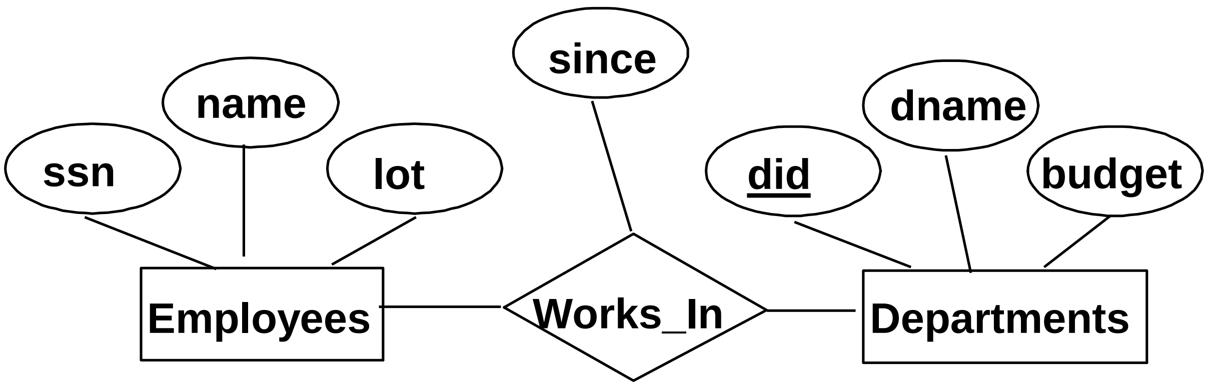
Possible Candidate Keys in Branch_Info table.

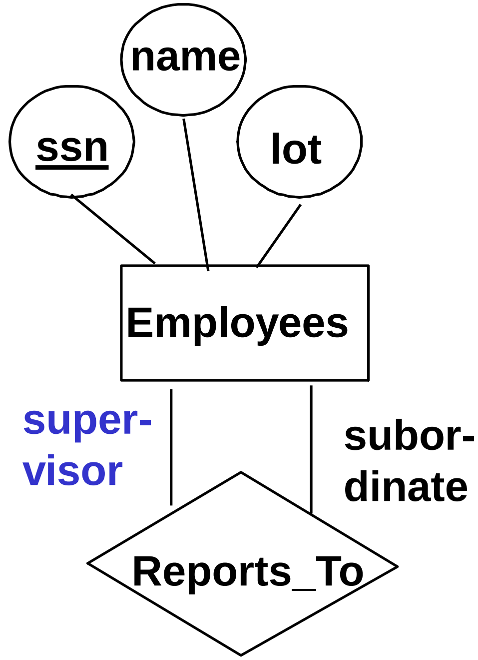
- Branch_Id
- Branch_Name
- Branch_Code

Among the above candidate keys one key can be primary key which uniquely identifies the entire relation. (Branch_Id), Alternate Key (Secondary Key) is Branch_name or Branch_code.

- Unique Key

ER Model Basics (Contd.)





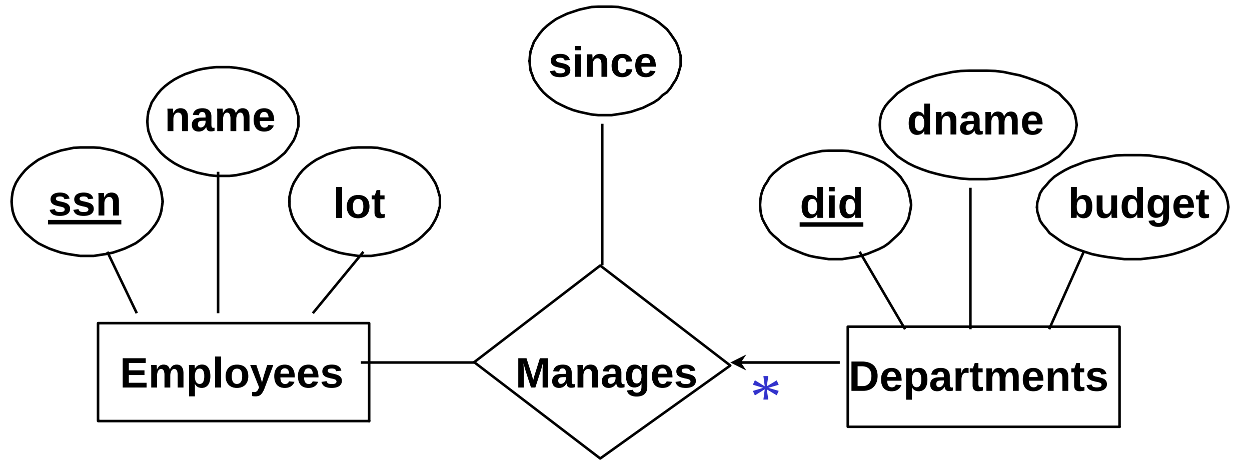
- *Relationship*: Association among two or more entities. E.g., Attishoo works in Pharmacy department.
- *Relationship Set*: Collection of similar relationships.
  - An n-ary relationship set R relates n entity sets E1 ... En; each relationship in R involves entities e1 E1, ..., en En
  - Same entity set could participate in different relationship sets, or in different "roles" in same set.

Key Constraints

- Consider Works_In: An employee can work in many departments; a dept can have many employees.

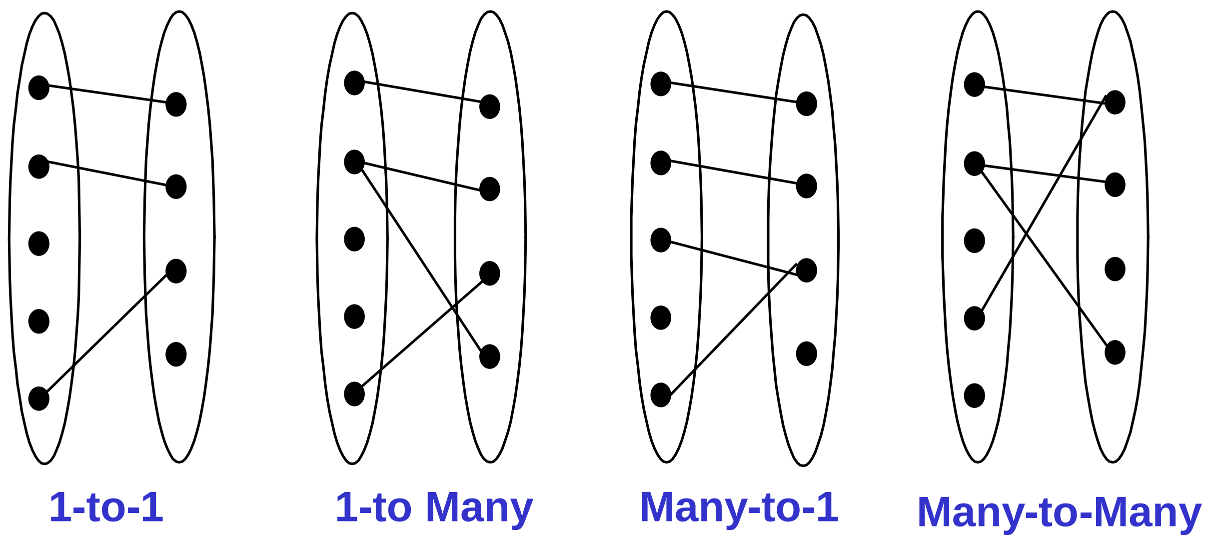
So many to many

Key Constraints



- In contrast, each dept has at most one manager, according to the *key constraint* on Manages.

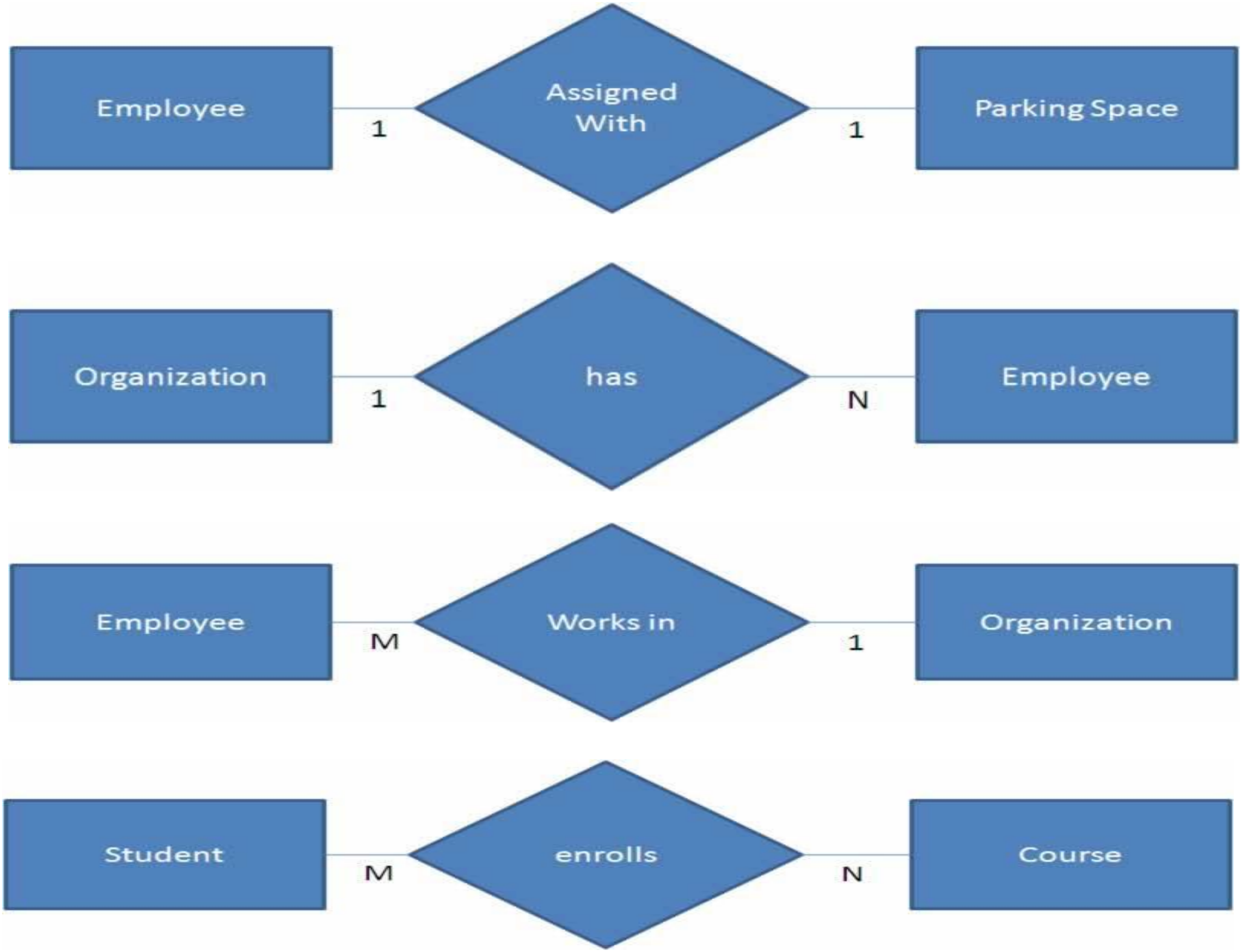
- * arrow indicates that given a dept it uniquely determines the Manages relationship in which it appears



Cardinality of a Relationship

- Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivity's as given below.
- 1. One to one (1:1) relationship
- 2. One to many (1:N) relationship
- 3. Many to one (M:1) relationship
- 4. Many to many (M:N) relationship

Examples



Employee
Assigned With
Parking Space
1
1
Organization
has
Employee
1
N
Employee
Works in
Organization
M
1
Student
enrolls
Course
M
N

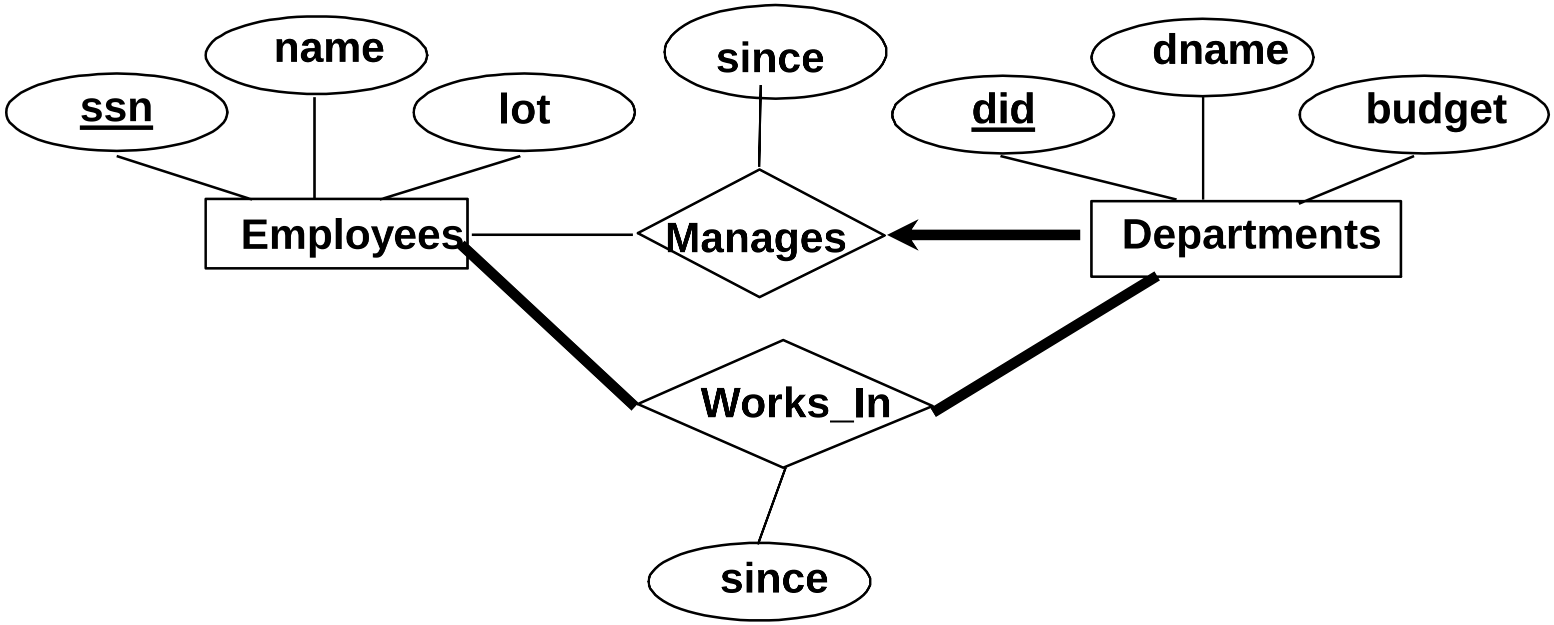
Degree of a Relationship

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

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

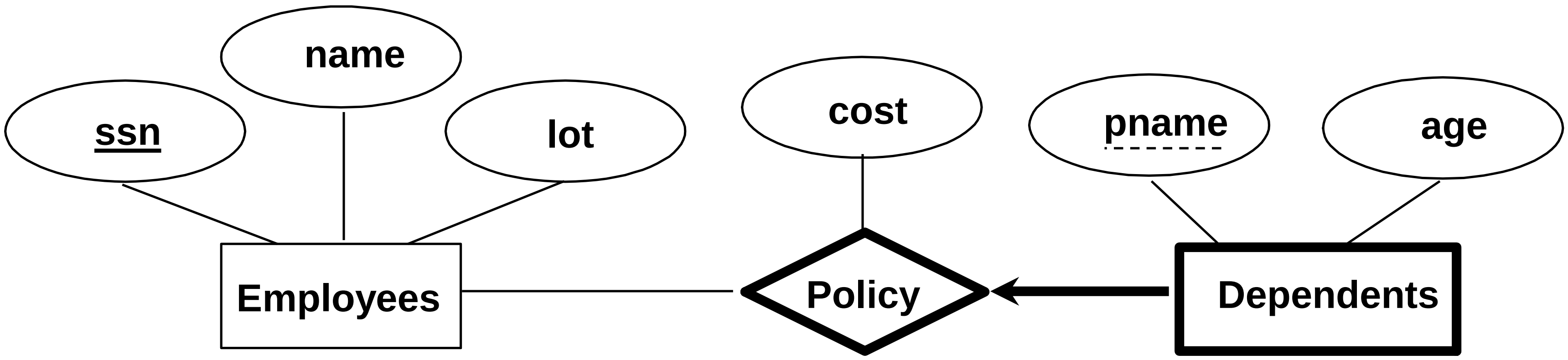
Participation Constraints

- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
- Every Department entity must appear in an instance of the relationship Works_In (have an employee) and every Employee must be in a Department
- Both Employees and Departments participate totally in Works_In

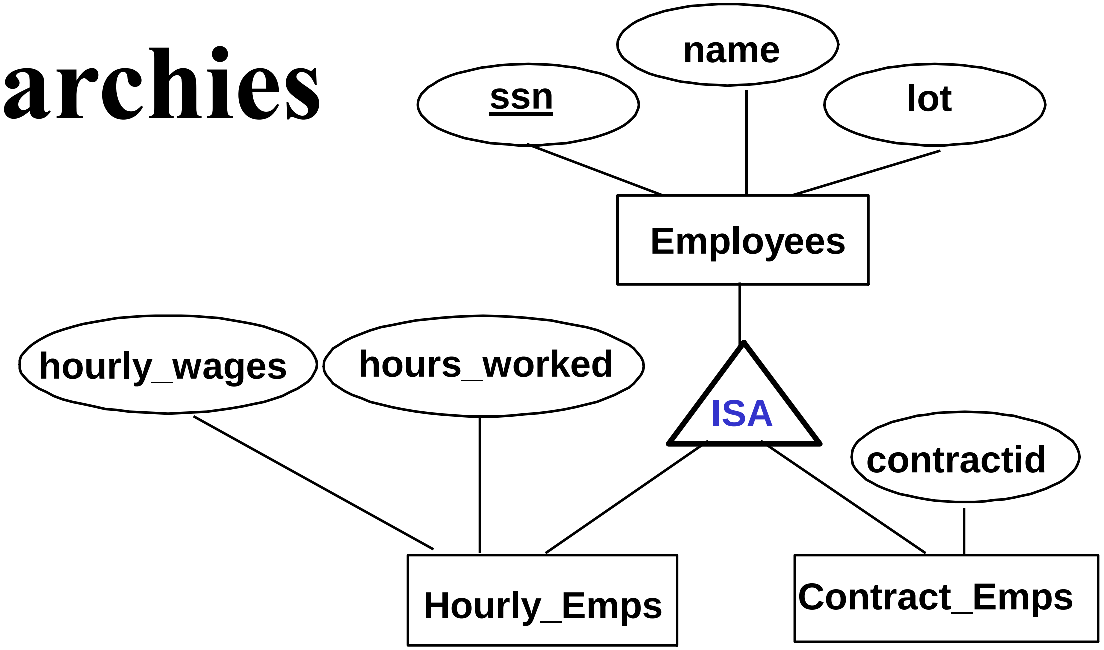


Weak Entities

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.



ISA (`is a') Hierarchies



❖ As in C++, attributes can be inherited.

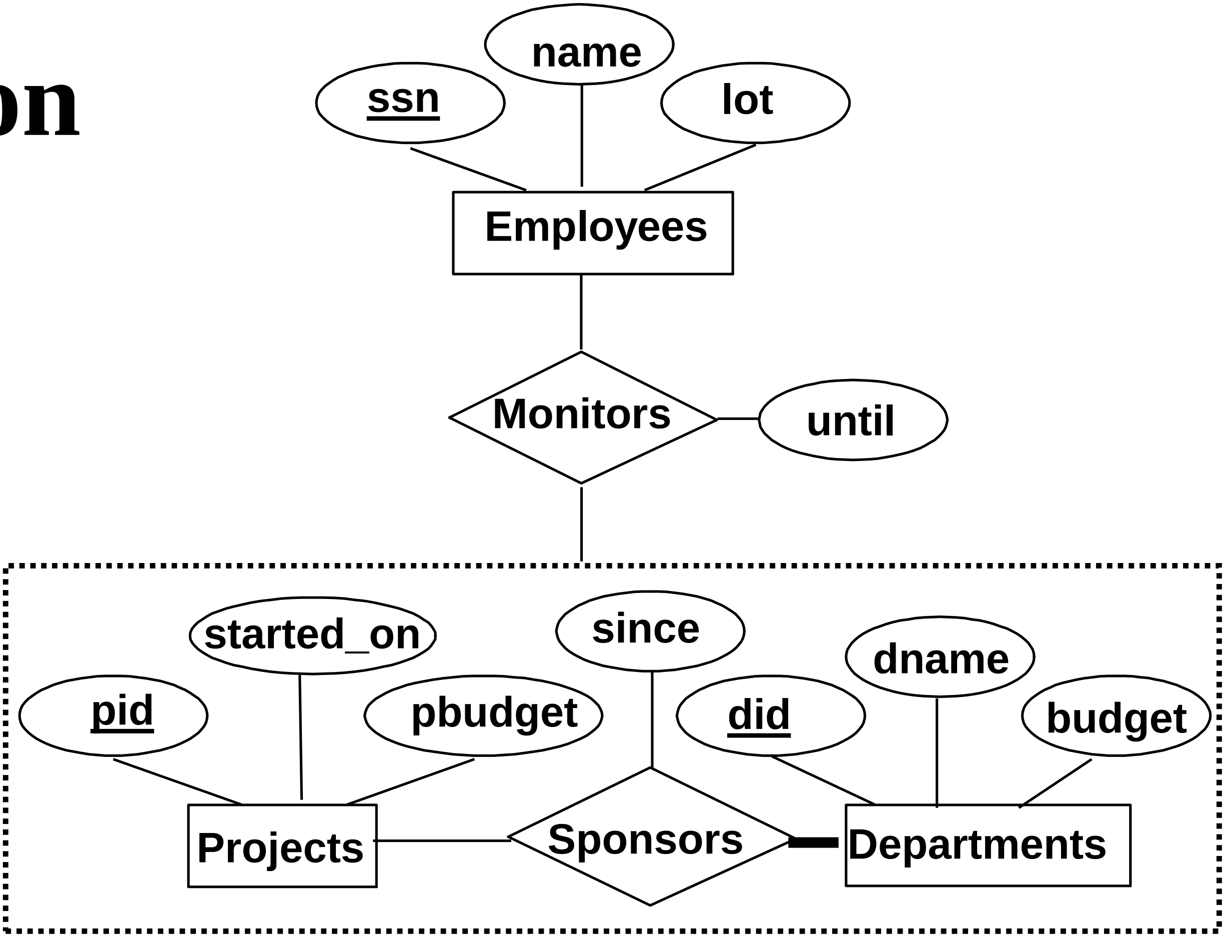
❖ If we declare A **ISA** B, every A entity is also considered to be a B entity.

❖Upwards is generalization. Down is specialization

Constraints in ISA relation

- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*
- Reasons for using ISA:
  - To add descriptive attributes specific to a subclass.
  - To identify entitities that participate in a relationship.

Aggregation



- Used when we have to model a relationship involving (entitity sets and) a *relationship set*.

Aggregation allows us to treat a relationship set as an entity set   for purposes of participation in (other) relationships.

Aggregation vs. ternary relationship:

- Monitors in last example is a distinct relationship, with a descriptive attribute.
- Also, can say that each sponsorship is monitored by at most one employee.

Conceptual Design Using the ER Model

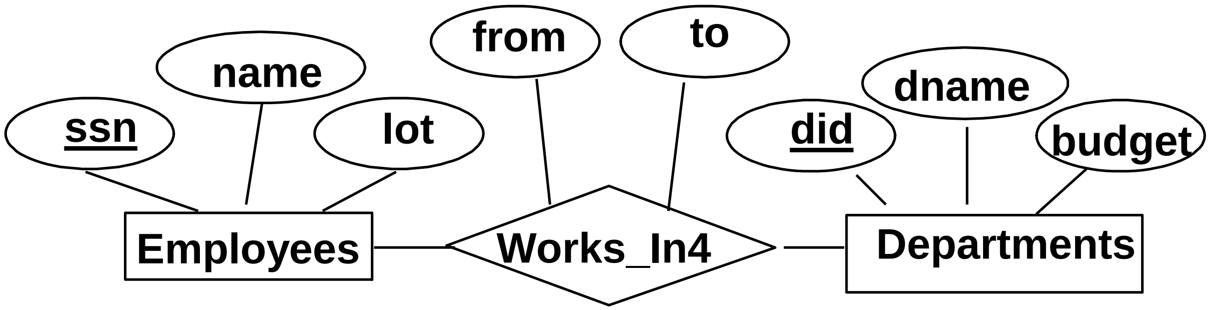
- Design choices:
  - Should a concept be modeled as an entity or a relationship?
  - Identifying relationships: Binary or ternary? Aggregation?
- Constraints in the ER Model:
  - A lot of data semantics can (and should) be captured.
  - But some constraints cannot be captured in ER diagrams.

Entity vs. Attribute

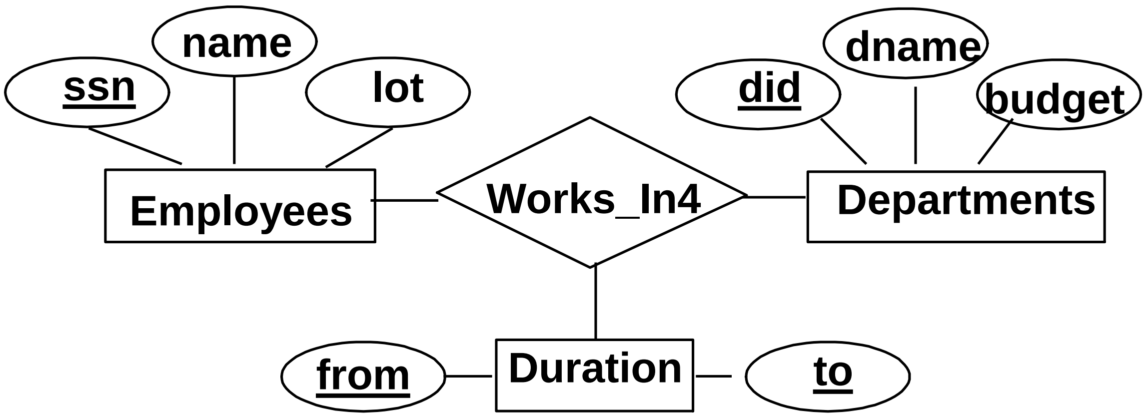
- Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?
- Depends upon the use we want to make of address information, and the semantics of the data:
  - If we have several addresses per employee, *address* must be an entity (since attributes cannot be set-valued).
  - If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, *address* must be modeled as an entity (since attribute values are atomic).

Entity vs. Attribute (Contd.)

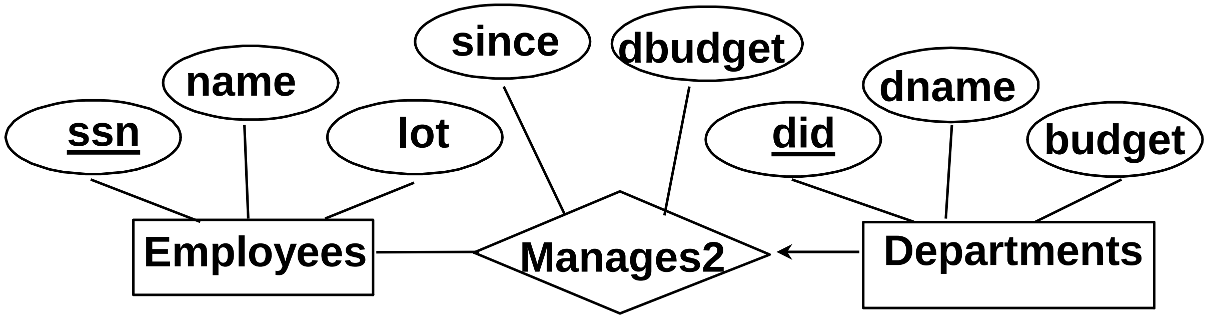
- Works_In4 does not allow an employee to work in a department for two or more periods.



- Similar to the problem of wanting to record several addresses for an employee: We want to record *several values of the descriptive attributes for each instance of this relationship.* Accomplished by introducing new entity set, Duration.

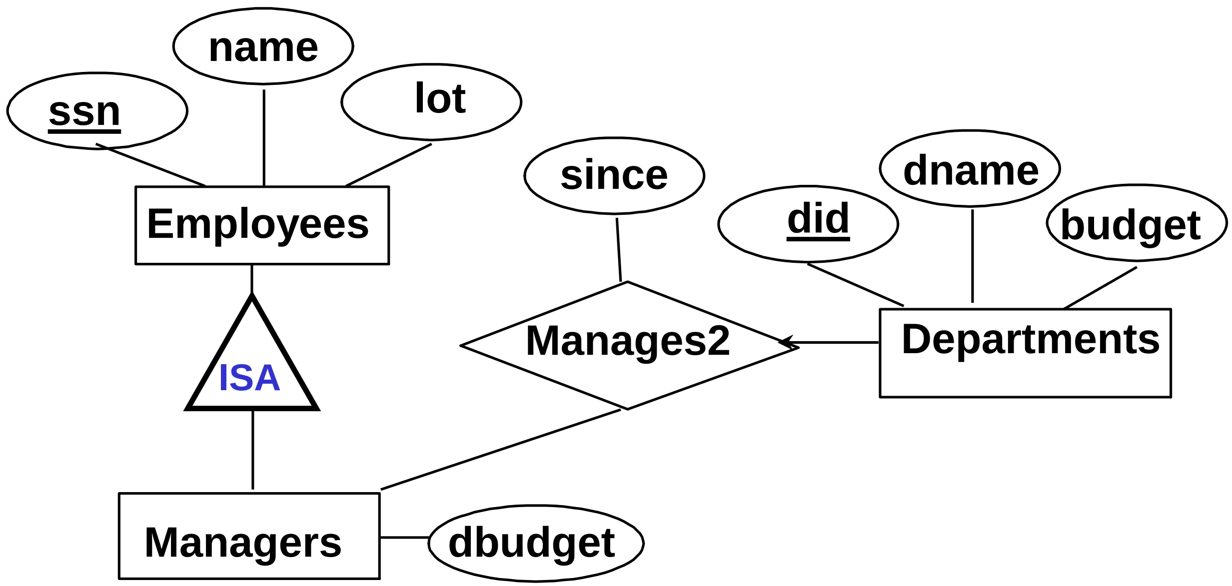


Entity vs. Relationship

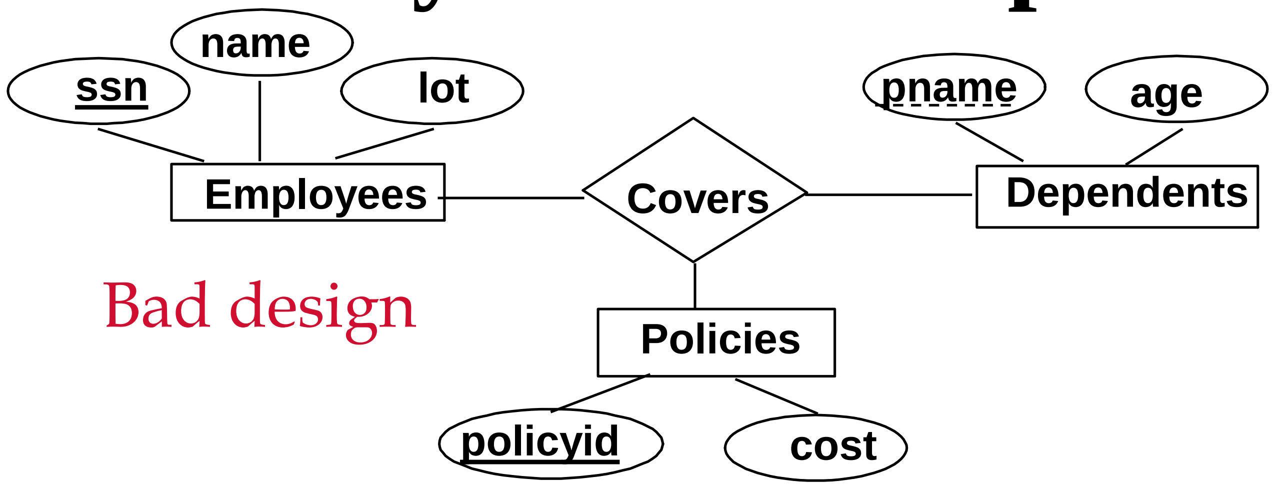


- ER diagram OK if a manager gets a separate discretionary budget for each dept.
- What if a manager gets a discretionary budget that covers *all* managed depts?
  - Redundancy: *dbudget* stored for each dept managed by manager.
  - Misleading: Suggests *dbudget* associated with department-mgr combination.

This fixes the problem!

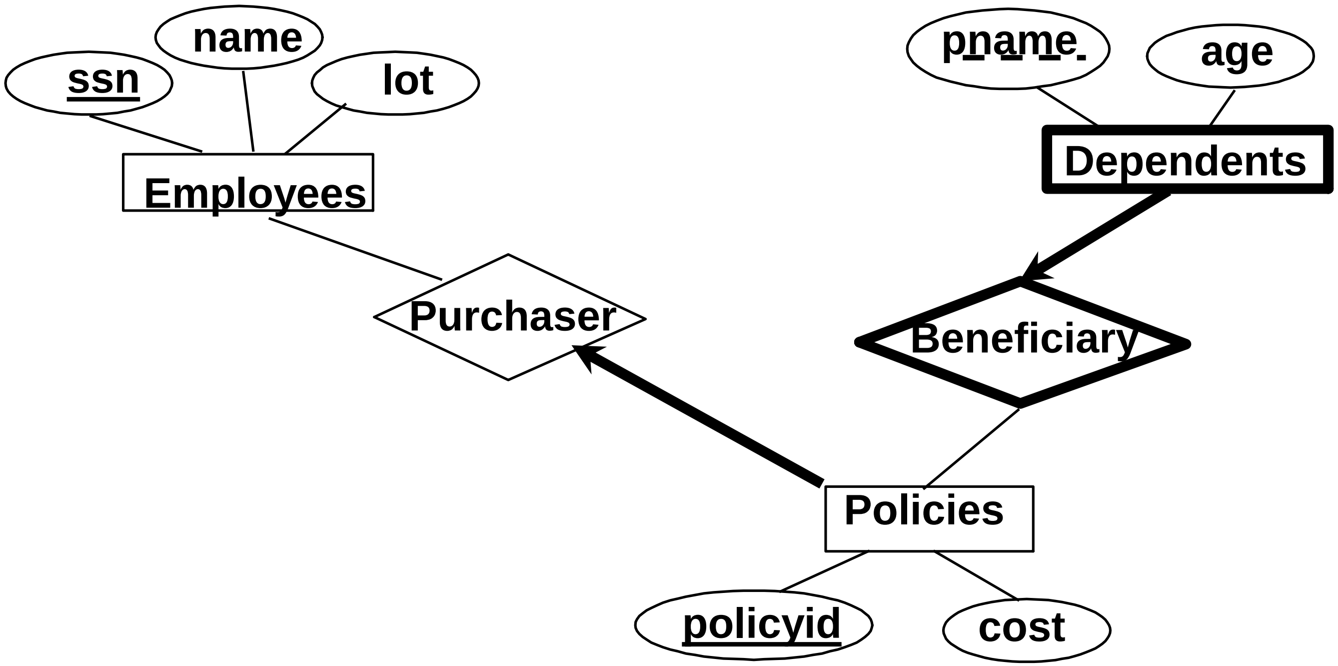


Binary vs. Ternary Relationships



- If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.
- What are the additional constraints do we need?

Better design



- Key constraint
- Total participation of policies in purchaser relationship

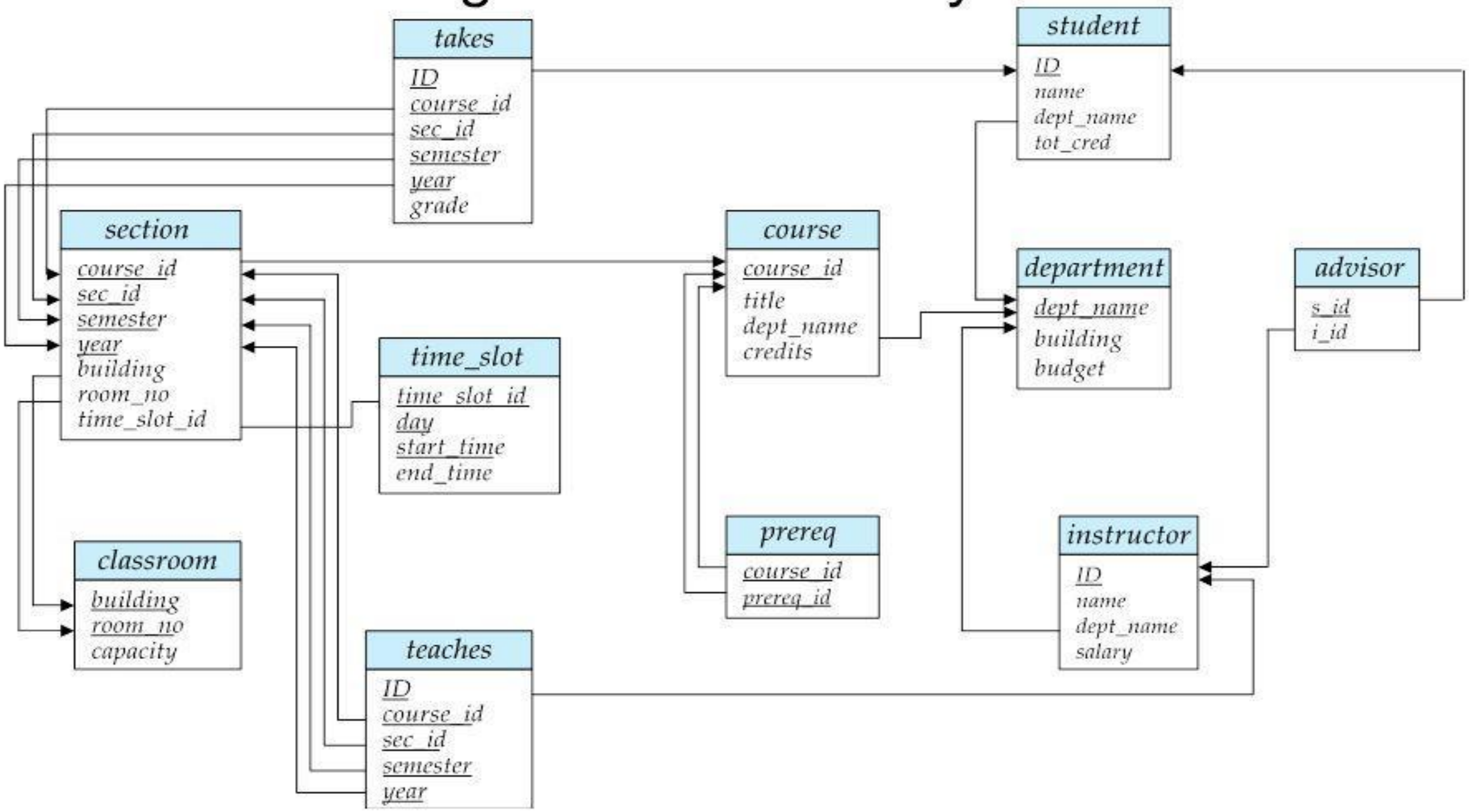
Binary vs. Ternary Relationships (Contd.)

- Previous example illustrated a case when two binary relationships were better than one ternary relationship.
- An example in the other direction: a ternary relation Contracts relates entity sets Parts, Departments and Suppliers, and has descriptive attribute *qty*. No combination of binary relationships is an adequate substitute:
  - S “can-supply” P, D “needs” P, and D “deals-with” S does not imply that D has agreed to buy P from S.
  - How do we record *qty*?

Summary of Conceptual Design

- *Conceptual design* follows *requirements analysis*,
  - Yields a high-level description of data to be stored
- ER model popular for conceptual design
  - Constructs are expressive, close to the way people think about their applications.
- Basic constructs: *entities*, *relationships*, and *attributes* (of entities and relationships).
- Some additional constructs: *weak entities*, *ISA hierarchies*, and *aggregation*.
- Note: There are many variations on ER model.

Schema Diagram for University Database



Summary of ER (Contd.)

- Several kinds of integrity constraints can be expressed in the ER model: *key constraints*, *participation constraints*, and *overlap/covering constraints* for ISA hierarchies. Some *foreign key constraints* are also implicit in the definition of a relationship set.
  - Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model.
  - Constraints play an important role in determining the best database design for an enterprise.

Summary of ER (Contd.)

- ER design is *subjective*. There are often many ways to model a given scenario! Analyzing alternatives can be tricky, especially for a large enterprise. Common choices include:
  - Entity vs. attribute, entity vs. relationship, binary or n-ary relationship, whether or not to use ISA hierarchies, and whether or not to use aggregation.
- Ensuring good database design: resulting relational schema should be analyzed and refined further. FD information and normalization techniques are especially useful.