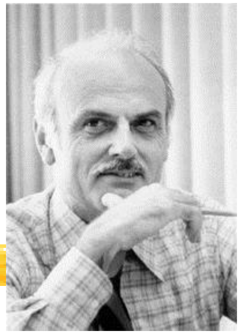
The Relational Model

Read Text Chapter 3

Learning Goals

- given an ER model of an application, design a minimum number of correct tables that capture the information in it
- given an ER model with inheritance relations, weak entities and aggregations, design the right tables for it
- given a table design, create correct tables for this design in SQL, including primary and foreign key constraints
- compare different table designs for the same problem, identify errors and provide corrections

Historical Perspective



- Introduced by Edgar Codd (IBM) in 1970
- Most widely used model today.
  - Vendors: IBM, Informix, Microsoft, Oracle, Sybase, etc.
- “Legacy systems” are usually hierarchical or network models (i.e., not relational)
  - e.g., IMS, IDMS, …

Historical Perspective

- Competitor: object-oriented model
  - ObjectStore, Versant, Ontos
  - A synthesis emerging: *object-relational model*
    - Informix Universal Server, UniSQL, O2, Oracle, DB2
- Recent competitor: XML data model
- In all cases, relational systems have been extended to support additional features, e.g., objects, XML, text, images, …

Main Characteristics of the Relational Model

- Exceedingly simple to understand
  - All kinds of data abstracted and represented as a table
- Simple query language ***separate*** from application language
- Lots of bells and whistles to do complicated things

Structure of Relational Databases

- ***Relational database:*** a set of ***relations***
- ***Relation:*** made up of 2 parts:
  - ***Schema*** : specifies name of relation, plus name and **domain** (type) of each **field** (or **column** or **attribute**).
    - e.g., Student (*sid*: string, *name*: string, *address*: string, *phone*: string, *major*: string).
  - ***Instance*** : a ***table,*** with rows and columns.
    #Rows = cardinality,
  - ***#fields = dimension / arity / degree***

Structure of Relational Databases

- Can think of a relation as a ***set*** of rows or ***tuples*** (i.e., all rows are distinct)
- ***Relational Database Schema:*** collection of schemas in the database
- ***Database Instance:*** a collection of instances of its relations

Example of a Relation Instance

attribute/
column name/
field

relation
name

Student

sid	name	address	phone	major
99111120	G. Jones	1234 W. 12th Ave., Van.	889-4444	CPSC
92001200	G. Smith	2020 E. 18th St., Van	409-2222	MATH
94001020	A. Smith	2020 E. 18th St., Van	222-2222	CPSC
94001150	Campeau J.	null	null	null

tuple/
row/
record

domain
value

column/
active
domain of
attribute
address

- Cardinality = 4, degree/arity = 5, all rows distinct
- Order of rows isn't important
- Order of fields – logically speaking not important, but may be important for some query languages

Formal Structure

- Formally, an n-ary relation r is a set of n-tuples
$$(a_1, a_2, \dots, a_n)$$
where a_i is in D_i , the domain (set of allowed values) of the i-th attribute.
- Attribute values are atomic
  - i.e., integers, floats, strings.

An entry cannot be a set or a tuple.

- Each attribute domain contains a special value ***null*** indicating that the value is not known.

Formal Structure

- If A_1 , ..., A_n are attributes with domains $D_1, \ldots D_n$, then
  $(A_1: D_1, \ldots, A_n: D_n)$ or $R(A_1: D_1, \ldots, A_n: D_n)$
  is a ***relation schema*** that defines a relation type
- $r(R)$ is a relation over the schema R (or, of type R).
- Formally, $r \subseteq D_1 \times \cdots \times D_n$, i.e., r is a subset of the cross product of its attribute domains.

Example of a formal definition

What if **address** had to be structured into **no**, **street** and **city?**

Student

sid	name	address	phone	major
99111120	G. Jones	1234 W. 12th Ave., Van.	889-4444	CPSC
92001200	G. Smith	2020 E. 18th St., Van	409-2222	MATH
94001020	A. Smith	2020 E. 18th St., Van	222-2222	CPSC
94001150	S. Wang	null	null	null

Student(sid: integer, name: string, address: string, phone: string, major: string)

Or, without the domains:

Student (sid, name, address, phone, major)

Relational Query Languages

- A major strength of the relational model: supports simple, powerful *querying* of data.
- Queries can be written intuitively, focusing on the ***what*** and the DBMS is responsible for efficient evaluation, i.e., the ***how***.
  - Precise semantics for relational queries.
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.

The SQL Query Language

- Developed by IBM (System R) in the 1970s
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision, current standard)
  - SQL-99 (major extensions)

The SQL Query Language

Student

sid	name	address	phone	major
99111120	**G. Jones**	1234 W. 12th Ave., Van.	**889-4444**	**CPSC**
92001200	G. Smith	2020 E. 18th St., Van	409-2222	MATH
94001020	**A. Smith**	2020 E. 18th St., Van	**222-2222**	**CPSC**

- To find the id's, names and phones of all CPSC students, we can write:

```
SELECT  sid, name, phone
FROM     Student
WHERE  major="CPSC"
```

sid	name	phone
99111120	G. Jones	889-4444
94001020	A. Smith	222-2222

- To select whole rows , replace "SELECT sid, name, phone " with "SELECT * "

Querying Multiple Tables

Student

sid	name	address	phone	major
99111120	G. Jones	…	…	CPSC
…	…	….	…	…

Grade

sid	dept	course#	mark
99111120	CPSC	122	80
…	…	….	…

- To select id and names of the students who have taken some CPSC course, we write:

```
SELECT sid, name
FROM Student, Grade
WHERE Student.sid = Grade.sid AND dept = 'CPSC'
```

- Note: This query "joins" two tables.
- Need to create/define and populate tables first though…

Creating Relations in SQL/DDL

- The statement on right
  - creates Student relation
  - observe that the type (domain) of each attribute is specified, and is enforced by the DBMS whenever tuples are added or modified

Just table name, its attributes and their types.

```
CREATE TABLE Student
    (sid      CHAR(8),
     name     CHAR(20),
     address  CHAR(30),
     phone    CHAR(13),
     major    CHAR(4))
```

- The statement on right
  - creates Grade relation
  - information about courses that a student takes

```
CREATE TABLE Grade
    (sid      CHAR(8),
     dept     CHAR(4),
     course#  CHAR(3),
     mark     INTEGER)
```

Destroying and Altering Relations

```
DROP TABLE Student
```

- Destroys the relation Student. The schema information *and* the tuples are deleted.

```
ALTER TABLE Student
      ADD COLUMN gpa REAL;
```

- The schema of Students is altered by adding a new attribute; every tuple in the current instance is extended with a ***null*** value for the new attribute. Why null?

Adding and Deleting Tuples

- Can insert a single tuple using:

```
INSERT
INTO      Student (sid, name, address, phone, major)
VALUES  ('52033688', 'G. Chan', '1235 W. 33, Van',
         '604-882-4444',  'PHYS')
```

- Can delete all tuples satisfying some condition (e.g., name = 'Smith'):

```
DELETE
FROM     Student
WHERE   name = 'Smith'
```

⌧ *Powerful variants of these commands are available; more later*

Integrity Constraints (ICs)

- **IC:** condition that must be true for *any* instance of the database; e.g., *domain constraints*
  - ICs are specified when schema is defined
  - ICs are checked when relations are modified
- A *legal* instance of a relation is one that satisfies all specified ICs
  - DBMS should not allow illegal instances
  - Avoids data entry errors, too!
- The kinds of ICs depend of the data model.
  - Our focus: ICs for relational databases

ICs

- We will several different kinds of ICs for relational DBs, including
  - Domain constraints: e.g., **name** must belong to **char(40); salary** must be float.
  - Key constraints: same meaning as in ER model.
  - Functional Dependencies: generalization of key constraints.
  - Foreign Key constraints.
- Ad hoc constraints can also be imposed: e.g., **salary** cannot be negative.

Keys Constraints (for Relations)

- Similar to those for entity sets in the ER model
- A set $S = \{A_1, A_2, \dots, A_m\}$ of attributes in an n-ary relation $(1 \leq m \leq n)$ is a ***key*** (or ***candidate key***) for the relation if :
  1. No two distinct tuples can have the same values in all key attributes, and
  2. No subset of S is itself a key (according to (1)).

  (If such a subset exists, then S is a *superkey* and not a key.)
- One of the possible keys is chosen (by the DBA) to be the ***primary key***.

A key is any minimal superkey.

- e.g.
  - {*sid*, *gpa*} is a superkey
  - ***sid*** is a key and the primary key for Students

minimal $-/\rightarrow$ minimum.

Keys Constraints in SQL

- Candidate keys are specified using the **UNIQUE** constraint
  - values for a group of fields must be unique (if they are not null)
  - these fields can be *null*
- A **PRIMARY KEY** constraint specifies a table's primary key
  - values for primary key must be unique
  - primary key attributes cannot be *null*
- Key constraints are checked when
  - new values are inserted
  - values are modified

A “Key” Note – The Big Picture

- A table has several keys, in general. – called candidate keys.
- One of them is declared to be the primary key.
- All candidate keys, whenever not null, must be unique.
- Primary key cannot ever be null.
- Where do all these keys come from? IOW, how can we tell which are the (candidate) keys of a table?
    - For now, just based on intuition and our understanding of an application and of the meaning of attributes.
    - In reality, (some) design specs are translated into special integrity constraints called ***functional dependencies***, from which keys are derived.

Key Constraints in SQL (contd.)

- (Ex.1- Normal) "For a given student and course, there is a single grade."

```
CREATE TABLE Grade
  (sid CHAR(8)
   dept  CHAR(4), course# CHAR(3),
   mark INTEGER,
   PRIMARY KEY  (sid,dept,course#) )
```

vs. This time, not just attributes & types, but also (key) constraints.

- (Ex.2 - Silly) "Students can take a course once, and receive a single grade for that course; further, no two students in a course receive the same grade."

```
CREATE TABLE Grade2
  (sid CHAR(8)
   dept  CHAR(4), course# CHAR(3),
   mark CHAR(2),
   PRIMARY KEY (sid,dept,course#),
   UNIQUE (dept,course#,mark) )
```

Foreign Keys (or Referential Integrity) Constraints

- ***Foreign key*** : Set of fields in one relation that is used to 'reference' a tuple in another (or same) relation.
  - Must correspond to the primary key of the other relation.
  - Like a 'logical pointer'.
  - **Note:** There are no physical pointers in the relational model.
- e.g.:
  Grade(*sid* string, *dept* string, *course#* string, *mark* integer)
  - *sid* is a foreign key referring to Student:
  - (*dept*, *course#*) is a foreign key referring to Course
- *Referential integrity*: All foreign keys reference existing entities.
  - i.e. there are no dangling references
  - all foreign key constraints are enforced,.
    - e.g, of a "data model" w/o RICs: html!

Some Common Examples of Referential Integrity

Crux: You can't refer to something that doesn't exist.
∀entity, there must be a "home table" where you'd look it up to see if it existed.

Some Examples

You can't rate a song that doesn't exist, i.e., is not listed in the Songs table, say.

There can't be a grade for something that is not a course nor for someone who isn't a registered bona fide student.

You can't borrow something from a library that is not a publication, nor return it.

List ***your*** examples here.

Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to have grades for courses.

```
CREATE TABLE Grade
   (sid CHAR(8), dept CHAR(4), course# CHAR(3), mark INTEGER,
    PRIMARY KEY (sid,dept,course#),
    FOREIGN KEY (sid) REFERENCES Student,
    FOREIGN KEY (dept, course#) REFERENCES Course )
```

Student

sid	name	address	Phone	major
53666	G. Jones	….	…	…
53688	J. Smith	….	…	…
53650	G. Smith	….	…	…

Grade

sid	dept	course#	mark
53666	CPSC	101	80
53666	RELG	100	45
53650	MATH	200	null
53666	HIST	201	60

Enforcing Referential Integrity

- Consider Students and Grade; *sid* in Grade is a foreign key that references Student.
- What should be done if a Grade tuple with a non-existent student id is inserted? (*Reject it!*)
- What should be done if a **Student tuple** is deleted?
  - Also delete all Grade tuples that refer to it?
  - Disallow deletion of this particular Student tuple?
  - Set sid in Grade tuples that refer to it, to a *default sid*.
  - Set sid in Grade tuples that refer to it, to *null*, (the special value denoting *`unknown'* or *`inapplicable'*.)
    - problem if sid is part of the primary key
- Similar if primary key of a Student tuple is updated

Referential Integrity in SQL/92

- SQL/92 supports all 4 options on deletes and updates.
  - Default is NO ACTION (*delete/update is rejected*)
  - **CASCADE** (also updates/deletes all tuples that refer to the updated/deleted tuple)
  - **SET NULL / SET DEFAULT** (referencing tuple value is set to the default foreign key value)

```
CREATE TABLE Grade
  (sid CHAR(8), dept CHAR(4),
   course# CHAR(3), mark INTEGER,
  PRIMARY KEY (sid,dept,course#),
  FOREIGN KEY (sid)
      REFERENCES Student
       ON DELETE CASCADE
       ON UPDATE CASCADE
  FOREIGN KEY (dept, course#)
      REFERENCES Course
       ON DELETE SET DEFAULT
       ON UPDATE CASCADE );
```

More ICs in SQL

Which among EGDs and TGDs do Key constraints and RICs resemble?

- SQL supports two types of IC's
  - table constraints
  - assertions
- Table constraints
  - are part of the table definition
  - domain, key, referential constraints, etc
  - Checked when a table is created or modified
- Assertions
  - Involve more than one table
  - Checked when any of the tables is modified
- SQL constraints can be named
  - CONSTRAINT studentKey PRIMARY KEY (sid)
- Constraints can be set to be
  - DEFERRED (applied at the end of the transaction
  - IMMEDIATE (applied at the end of a SQL statement)
- More on constraints when we discuss SQL in depth.

Where do ICs Come From?

- ICs are based upon the semantics of the real-world enterprise being described (using the database relations).
- We can check a database instance to verify a ***given*** IC, but we cannot tell what the ICs are just by looking at the instance.
  - For example, even if all student names differ, we cannot assume that name is a key.
  - An IC is a statement about *all possible (i.e., legal) instances*.

Name	Phone	Address
John Doe	123-4567	123 Main St.
Peter Bloe	765-4321	765 Cambie St.
Jane Doe	234-5678	234 Oak St.

  - What if we added (`Jane Doe', 123-7654, `765 Fraser')?

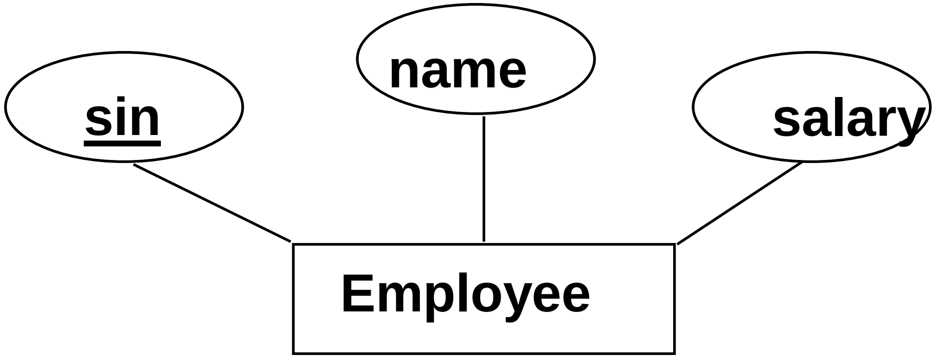
Where do ICs come from?

- **All constraints must be identified during the conceptual design.**
- Some constraints can be explicitly specified in the conceptual model
  - Primary Key and foreign key ICs are shown on ER diagrams.
- Others are written in a more general language.

Logical DB Design: ER to Relational

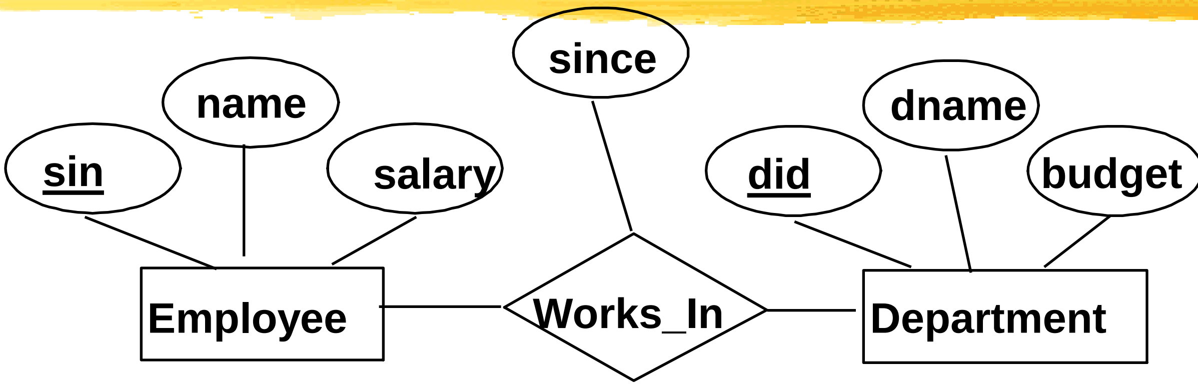
Entity sets to tables.

- Each strong entity set is mapped to a table.
  - entity attributes = table attributes
  - entity keys = table keys




```
CREATE TABLE Employee
    (sin    CHAR(11),
     name CHAR(20),
     salary    FLOAT,
     PRIMARY KEY  (sin))
```

Relationship Sets to Tables: Simple Case

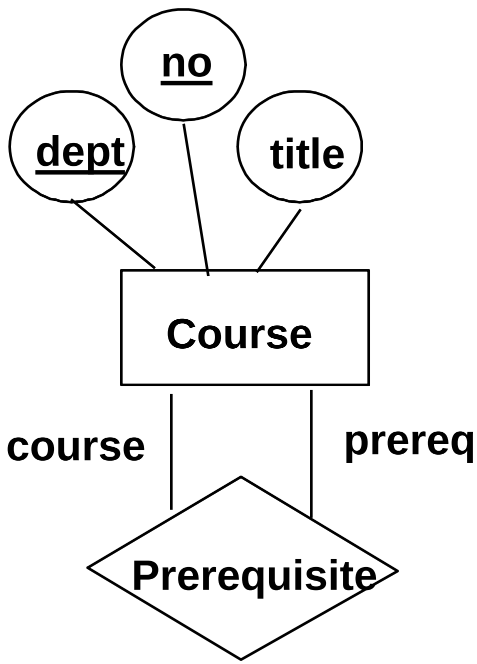


sin and did CANNOT be null

- Relationship set has no constraints (i.e., many-to-many partial)
- In this case, it is mapped to a new table.
- Attributes of the table must include:
  - Keys for each participating entity set as foreign keys.
    - This set of attributes forms a *key* for the relation.
  - All descriptive attributes.

```
CREATE TABLE Works_In(
  sin     CHAR(11),
  did     INTEGER,
  since   DATE,
  PRIMARY KEY (sin, did),
  FOREIGN KEY (sin)
      REFERENCES Employee,
  FOREIGN KEY (did)
      REFERENCES Department)
```

Relationship Sets to Tables (contd.)

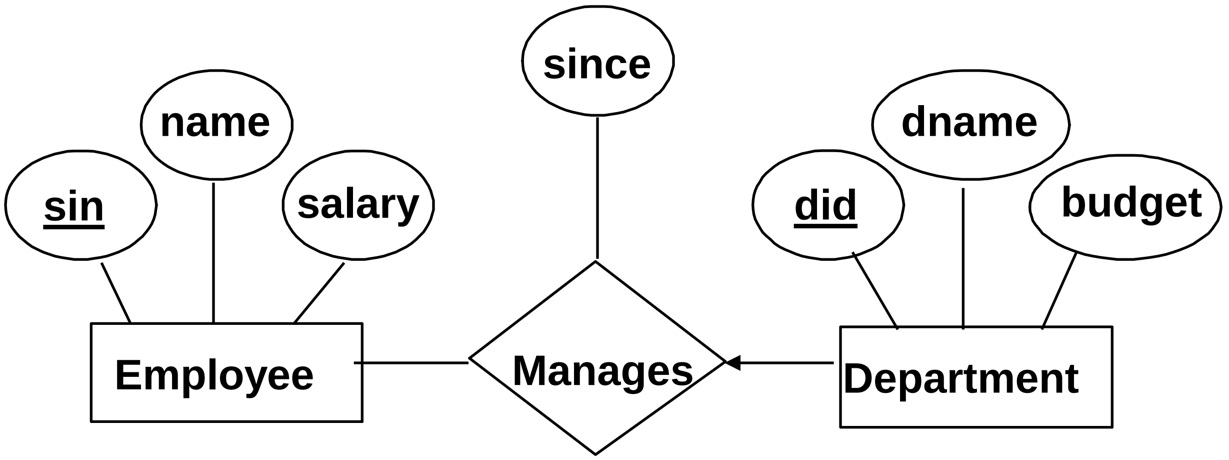


- In some cases, we need to use the roles:

```
CREATE TABLE Prerequisite(
  course_dept     CHAR(4),
  course_no     CHAR(3),
  prereq_dept     CHAR(4),
  prereq_no     CHAR(3),
  PRIMARY KEY (course_dept, course_no,
                    prereq_dept, prereq_no),
  FOREIGN KEY (course_dept, course_no)
     REFERENCES Course(dept, course#),
  FOREIGN KEY (prereq_dept, prereq_no)
     REFERENCES Course(dept, course#))
```

Pay attention to the roles in case of self-relationship sets.

Relationship Sets with Cardinality Constraints



- Each dept has at most one manager, according to the *key constraint* on Manages.
- How should we represent Manages?

Translating ER Diagrams with Key Constraints

- WRONG WAY:
  - Create a separate table for Manages:
  - Note that did is the key now!
  - Create separate tables for Employee and Department.

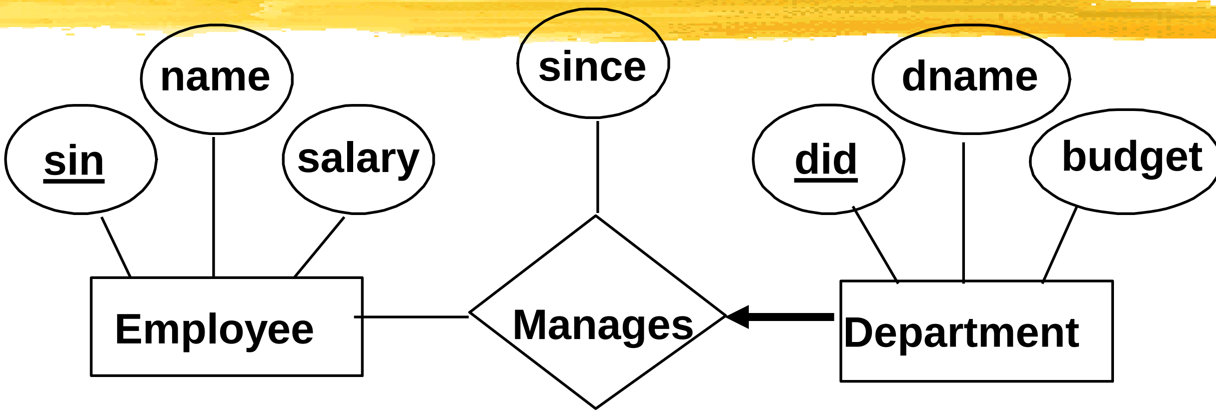
```
CREATE TABLE  Manages(
   sin      CHAR(11),
   did      INTEGER,
   since  DA
   PRIMARY
   FOREIGN KEY (sin) REFERENCES Employee,
   FOREIGN KEY (did) REFERENCES Department)
```

Qn: how can we state the participation constraint *Every dept must have a manager?*

- RIGHT WAY
  - Since each department has a unique manager, we can **combine Manages and Department into one table.**
  - Create another table for Employee

```
CREATE TABLE  Dept_Mgr(
   did       INTEGER,
   dname  CHAR(20),
   budget  REAL,
   sin        CHAR(11),
   since     DATE,
   PRIMARY KEY  (did),
   FOREIGN KEY (sin) REFERENCES Employee)
```

Translating Participation Constraints



- Every department must have a (unique) manager.
  - Every *did* value in Department table must appear in a row of the Manages table (with a non-null *sin* value)
- How can we express that in SQL?

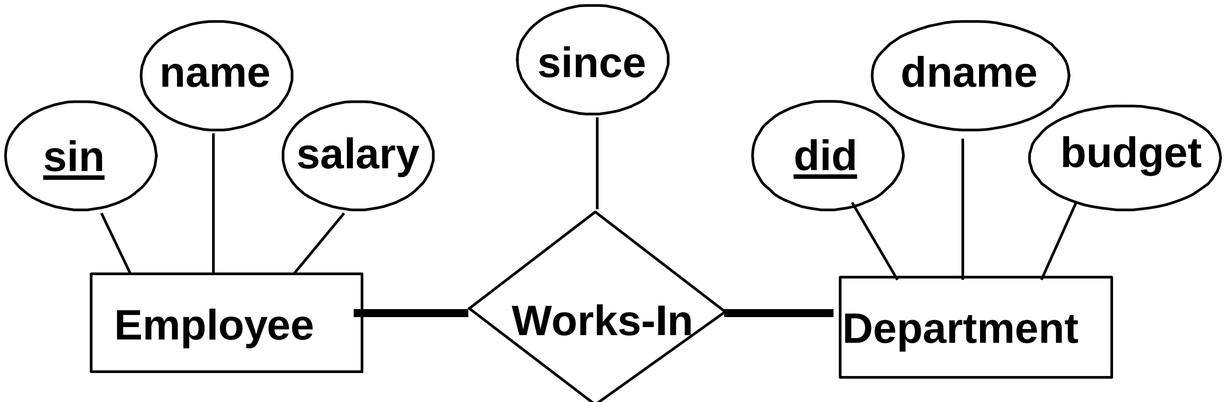
Participation Constraints in SQL

- Using the right method for Manages (add Manages relation in the Department table), we can capture participation constraints by
  - ensuring that each **did** is associated with a **sin** that is not null
  - not allowing to delete a manager before he/she is replaced

```
CREATE TABLE Dept_Mgr(
   did        INTEGER,
   dname      CHAR(20),
   budget     REAL,
   sin        CHAR(11) NOT NULL,
   since      DATE,
   PRIMARY KEY (did),
   FOREIGN KEY (sin) REFERENCES Employee,
          ON DELETE NO ACTION
          ON UPDATE CASCADE)
```

- **Note: We cannot express this easily if the wrong method was used for Manages.**

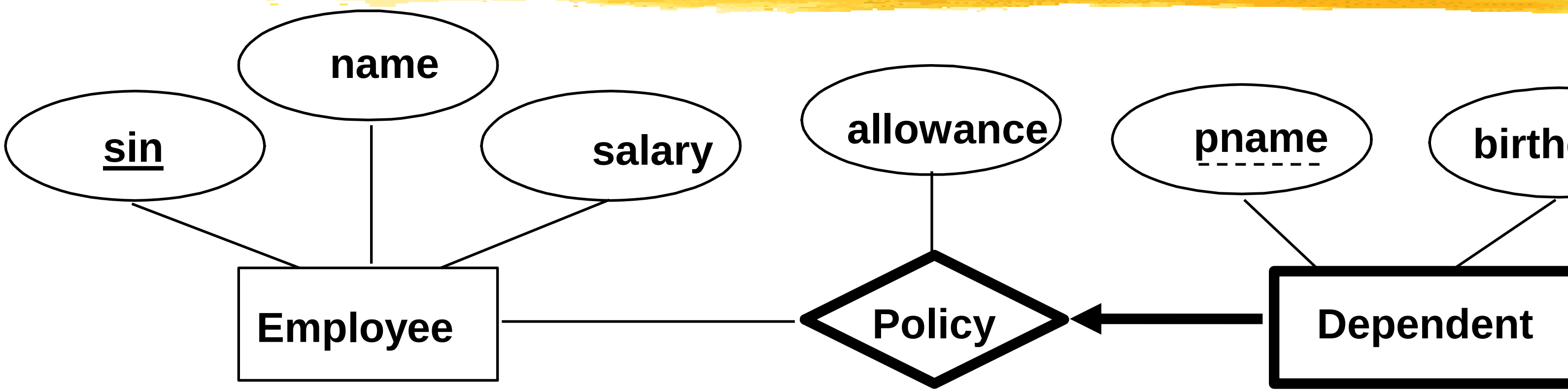
Participation Constraints in SQL (contd.)



$$Dept(Did, Dn, B) \rightarrow (\exists S, Sin) WI(Sin, Did, S).$$

- How can we express that
  “every employee works in one or more departments and every department has one or more employees in it”?
- Neither foreign-key nor not-null constraints in Works_In can do the job.
- We need assertions (to be discussed later)

Translating Weak Entity Sets



- A ***weak entity*** is identified by considering the primary key of the *owner* (strong) entity.
  - Owner entity set and weak entity set participates in a one-to-many identifying relationship set.
  - Weak entity set has total participation.
- What is the best way to translate it?

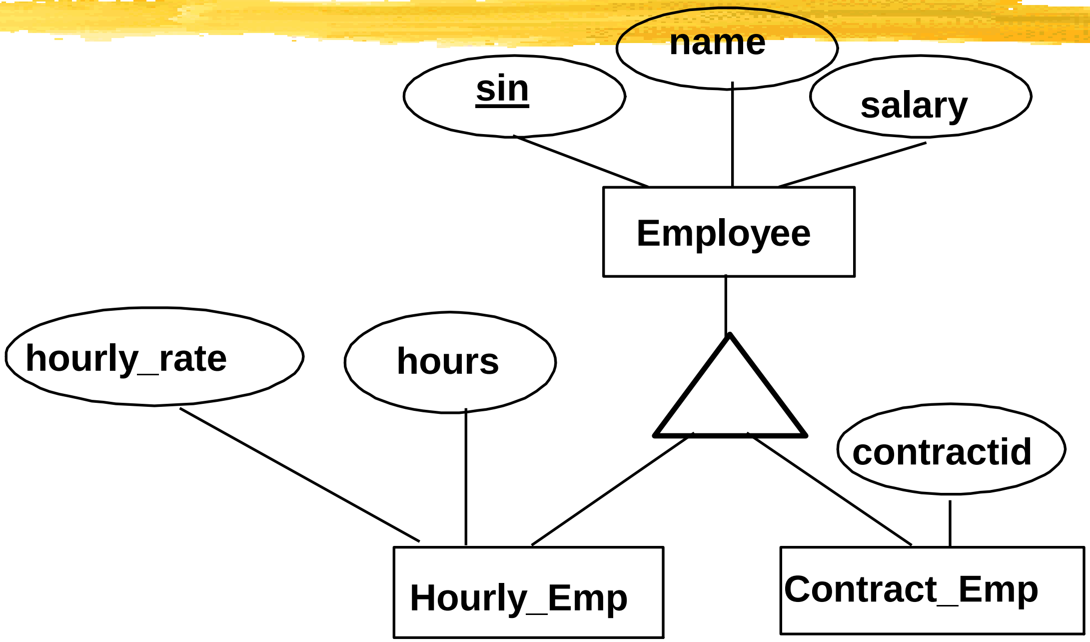
Translating Weak Entity Sets (contd.)

- Weak entity set and its identifying relationship set are translated into a single table.
  - Primary key would consist of the owner's primary key and week entity's partial key
  - When the owner entity is deleted, all owned weak entities must also be deleted.

```
CREATE TABLE Dependent (
  pname        CHAR(20),
  birthday     DATE,
  allowance    REAL,
  sin          CHAR(11) NOT NULL,
  PRIMARY KEY (sin, pname),
  FOREIGN KEY (sin) REFERENCES Employees,
                    ON DELETE CASCADE,
                    ON UPDATE CASCADE)
```

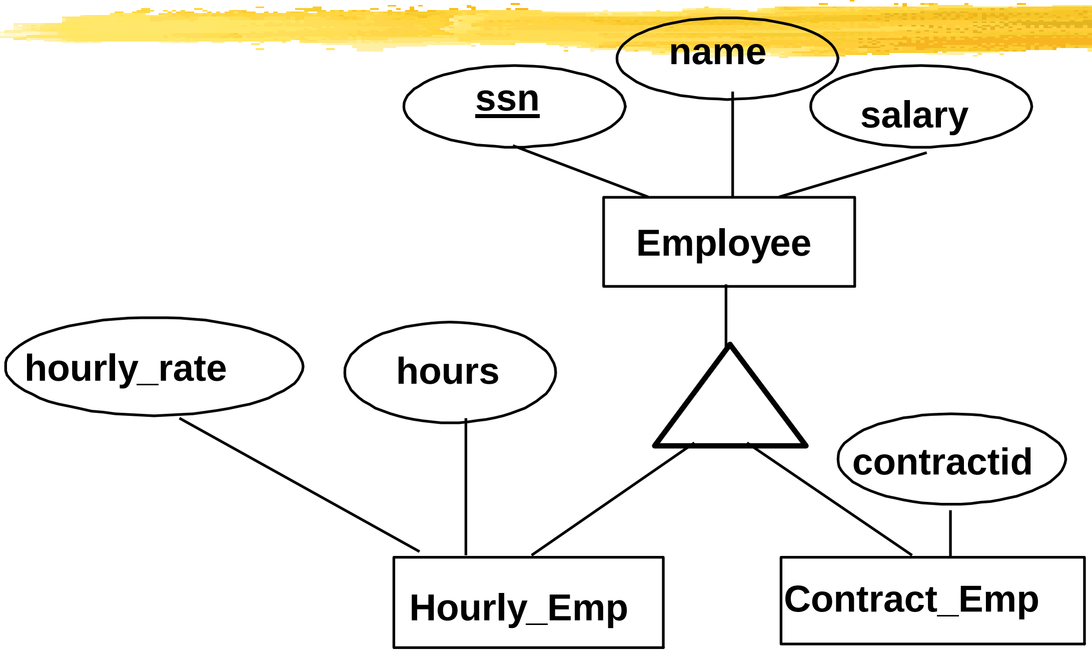
Could ON DELETE NO ACTION ever make sense for this case?

Translating ISA Hierarchies to Tables



- *What is the best way to translate this into tables?*

First Attempt : Totally unsatisfactory



- Safe but with lots of duplication
- Not a good answer!

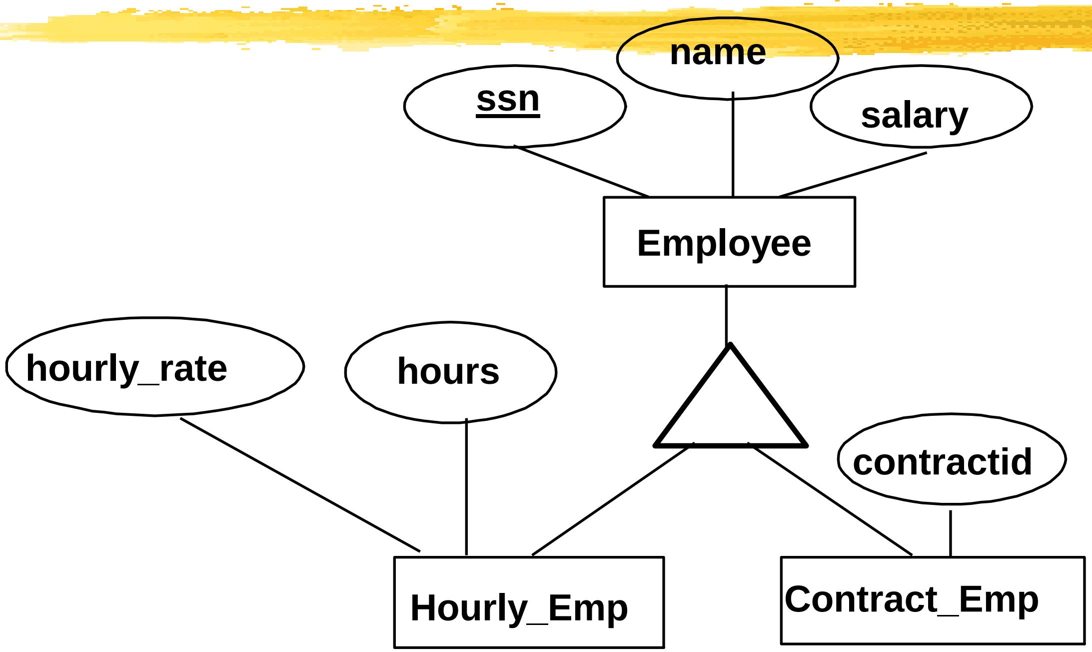
One table per entity. Each has *all* attributes:

Employee(sin, name, salary)

Hourly_Emp(sin, name, salary, hourly_rate, hours)

Contract_Emp(sin, name, salary, contractid)

Method 1 : One table



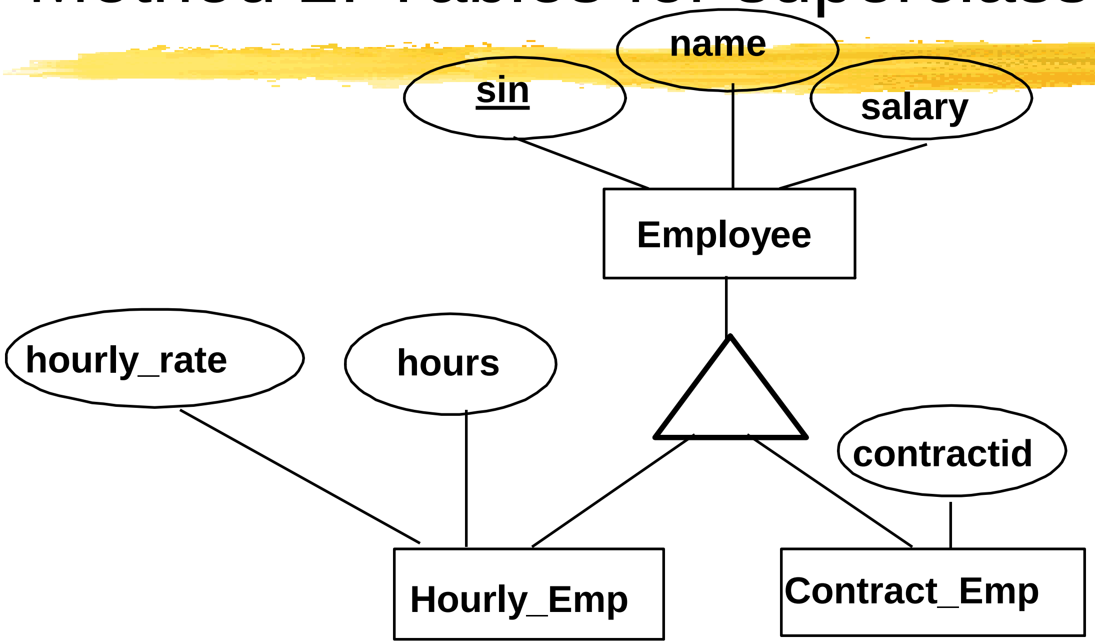
- Wastes a lot of space for null values
- Worse when there are many subclasses with many attributes
- Excellent method if subclasses do not have any new attributes and relationships

- One table which has all attributes and a *type* field which can have values to identify the type of the employee:

Employee(<u>sin</u>, name, salary, hourly_rate, hours, contractid, type)

If subclasses do not have special attrs of their own, was it necessary to model diff. categories of emp's as subclasses?

Method 2: Tables for superclass and subclasses



- Works well for queries involving only superclass attributes or only the extra subclass attributes
- Have to combine two tables to get all attributes for a subclass

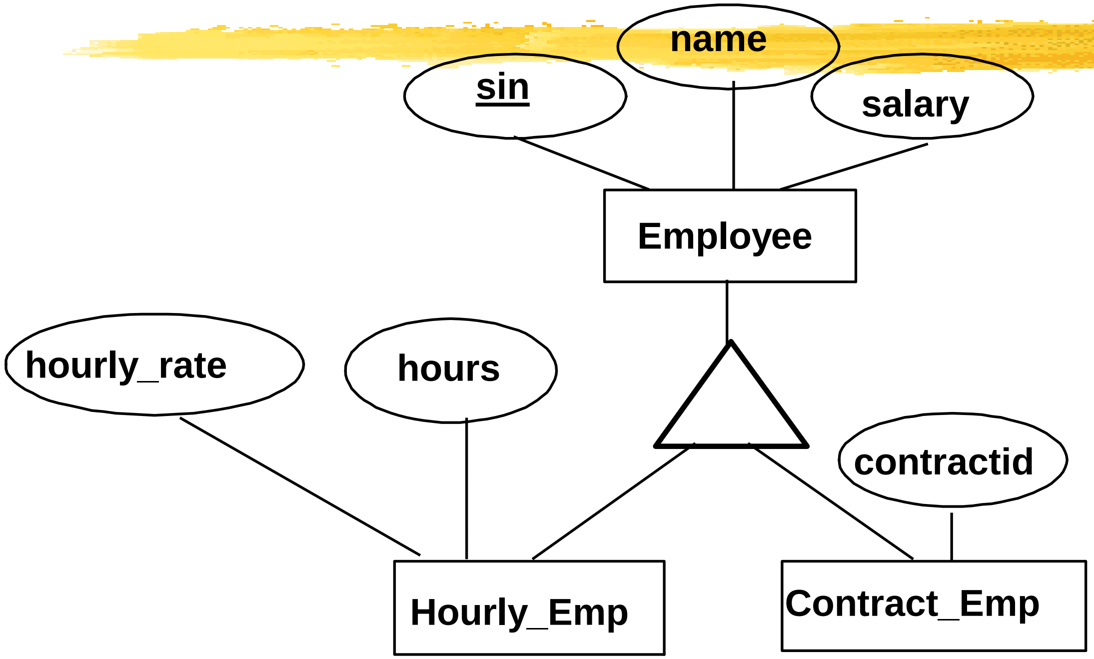
- Superclass table contains all superclass attributes
- Each subclass table contains primary key of superclass and the subclass attributes

  Employee(<u>sin</u>, name, salary)

  Hourly_Emp(<u>sin</u>, hourly_rate, hours)

  Contract_Emp(<u>sin</u>, contractid)

Method 3: Only subclass tables



- ❑ If ISA-relation is partial, it cannot be applied (may loose entities)
- ❑ Works poorly if the superclass participates in any relationship (should not be applied in that case)
- ❑ If ISA-relation is not disjoint, it duplicates info

- No table for superclass
- One table per subclass containing
  - *all* superclass and subclass attributes

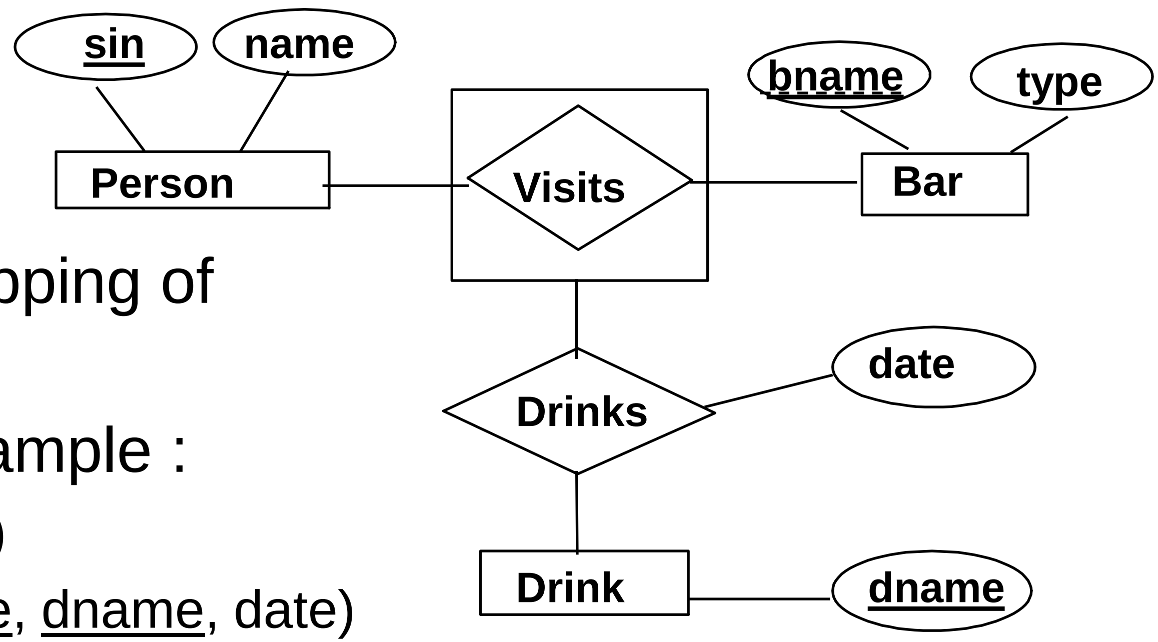
Hourly_Emp(<u>sin</u>, name, salary, hourly_rate, hours)

Contract_Emp(<u>sin</u>, name, salary, contractid)

Summary for Translating ISA to Tables

- **If subclasses have no new attributes or if they all have the same attributes and relationships**
  - One table with all attributes and an extra ***category*** attribute for the subclasses
  - If we need the subclasses, we can define them as views
- **If ISA is <u>total</u> and <u>disjoint</u>, subclasses have different attributes or different relationships and there is no need to keep the superclass**
  - No table for the superclass
  - One table for each subclass with all attributes
- **Otherwise (if ISA is partial or overlapping and subclasses have new and different attributes….)**
  - A table for the superclass and one for each subclass

Translating Aggregation



- Use standard mapping of relationship sets
- Tables for our example :
  - Visits(sin, bname)
  - Drinks(sin, bname, dname, date)
- Special Case:
  - If Drinks is total on Visits and Visits has no descriptive attributes we could keep only the Drinks table (discard Visits).

Views

- A view is just a virtual table, often defined using a query: ***Normally***, we only store the **view's definition**, rather than the rows in the view query's result.

```
CREATE VIEW CourseWithFails(dept, course#, title, mark) AS
        SELECT  C.dept, C.course#, title, mark
        FROM    Course C, Enrolled E
        WHERE   C.dept = E.dept AND
                C.course# = E.course# AND mark<50
```

- Views can be used as any other table.
- System usually evaluates them on the fly.
- Views can be dropped using the **DROP VIEW** command.

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Given CourseWithFails, but not Course or Enrolled, we can find the course in which some students failed, but we can't find which students failed.
  - What kind of users might such a view be appropriate for?
- Views can be used to define subsets of entities and relationships that have to satisfy certain constraints.

View Updates

- View updates must occur at the base tables.
  - Ambiguous
  - Difficult
- DBMS's restrict view updates only to some simple views on single tables (called updatable views)
- How to handle DROP TABLE if there's a view on the table?
- DROP TABLE command has options to prevent a table from being dropped if views are defined on it:
  - DROP TABLE Student RESTRICT
    - drops the table, unless there is a view on it
  - DROP TABLE Student CASCADE
    - drops the table, and recursively drops any view defined on it

Relational Model: Summary

- A tabular representation of data.
- Simple and intuitive, currently the most widely used.
- Integrity constraints can be specified, based on application semantics. DBMS checks for violations.
  - Important ICs: primary and foreign keys
  - Additional constraints can be defined with assertions (but are expensive to check)
- Powerful and natural query languages exist.
- Rules to translate ER to relational model
- SQL lets us specify ICs more general than just primary keys.
- SQL ICs are still restricted special cases of EGDs and TGDs.