The Relational Model

UVic C SC 370

Dr. Daniel M. German

Department of Computer Science

University of Victoria

May 19, 2003 Version: 1.1.1

Overview

- How is data represented in the relational model
- What integrity constrains can be expressed?
- How can data be created and modified?
- How can data be manipulated and queried?
- How can we create, modify, and query tables using SQL?
- How can we obtain a relational database from an ER diagram?

Chapter 3 of textbook

Relational Model

- Proposed by Codd in 1970
- By far the most dominant data model for databases
- It is a very simple and elegant model:
  - A database is a collection of one or more relations
  - Each relation is a table with rows and columns
- This is a very simple model for data
- That allows for a very powerful query and manipulation languages

Relation

- A **relation** consists of:
  - A **relation schema**: describes the **name** of the table, the **name** and **domain** of each field (or column, or attribute)

    $Students(sid : string, name : string, age : integer, gpa : real)$

  - A **relation instance**, an actual relation

sid	*name*	*login*	*age*	*gpa*
53666	Jones	Jonescs	18	7.4
53668	Smith	smithee	18	7.8
53650	Smith	smithmath	19	6.4
53831	Madayan	madayan@music	11	8.0
53832	Guldy	guldu@music	12	2.0

  - The order is irrelevant
  - In theory, no two rows can be identical

Domain Constraints

- The **domain constraints** indicate the valid set of values from which columns can take their value
- The **domain** is like a **type**
- Formally:

$$\{\langle f_1 : d_1, ..., f_n : d_n\rangle | d_1 \in Dom_1, ..., d_n \in Dom_n\}$$

- An instance of the relations students can be expressed:

$$\langle sid : 53666, name : Jones, login : Jones@cs, age : 18, gpa : 7.4\rangle$$

- More specifically, **a relation instance** means *a relation instance that satisfies the domain constraints of the schema*

More Terminology

- The **degree** or **arity** of a relation is its number of fields
- The **cardinality** of a relation is the number of tuples in it.
- A **relational database** is a collection of relations with distinct relation names
- An **instance of a relational database** is a collection of relation instances, one per relation schema in the database schema.

SQL

- In SQL a **table** denotes a relation
- We use the **Data Definition Language –DDL–** to create, modify or delete tables
- The command CREATE TABLE is used to define a table

```
CREATE TABLE students (
        sid     char (20),
        name    char (30),
        login   char(20),
        age     integer,
        gpa     real
);
```

- To add a tuple to the relation:

```
INSERT
INTO Students(sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2);
```

- To delete a tuple from the relation:

```
DELETE
FROM Students
WHERE Name = 'Smith';
```

Integrity Constraints over Relations

- An **integrity constraint (IC)** is a condition on the db schema that restricts the data that can be stored in an instance of the db
- An instance that satisfies all its ICs is said to be a **legal instance**
- ICs are specified at definition time
- ICs are enforced at run time

Key Constraints

- A **key constraint** is a statement that certain *minimal* subset of the fields of a relation is a unique identifier of the tuple
- A set of fields that uniquely identifies a tuple is called a **candidate key** or just a key:
  - Two distinct tuples of the relation cannot have the same values in all the fields of the key
  - No subset of the set of fields in a key is a unique identifier of the key
- A **superkey** is a set of fields that contains a **key**
- Question, *what are the risks of wrongly defining a key?*

Key Constraints...

- Every relation is guaranteed to have a key
- Out of all available keys, the DB designer should identify a **primary** key

Key Constraints in SQL

- We can declare a key using the `UNIQUE` constraint
- At most one of these *candidate* keys can be declared as a **primary key** using `PRIMARY KEY`

```
create table students (
        sid         char (20),
        name        char (30),
        login       char(20),
        age         integer,
        gpa         real,
        UNIQUE      (name, age),
        CONSTRAINT StudentsKey PRIMARY KEY (sid)
);
```

Foreign Key Constraints

- Sometimes the information stored in a relation is linked to the information stored in another one
- If one is modified, the other must be checked and potentially modified in order to keep the data consistent
- Example:

  Enrolled(*studid*: string, *cid*: string, *grade*: string)

- We need to insure that only students in the *Students* relationship can be added to the *Enrolled* one
- The *studid* field of *Enrolled* is called a **foreign key** and **refers** to *Students*.

Foreign Key Constraints

cid	*grade*	*studid*
Donuts 101	C	53831
Ballroom 203	B	53832
Topology 114	A	53650
History 105	B	53666

(a) Enrolled

sid	*name*	*login*	*age*	*gpa*
53666	Jones	Jonescs	18	7.4
53668	Smith	smithee	18	7.8
53650	Smith	smithmath	19	6.4
53831	Madayan	madayan@music	11	8.0
53832	Guldy	guldu@music	12	2.0

(b) Students

- If we try to insert $\langle 55555, Art104, A \rangle$ into Enrolled, the IC is violated (it should be rejected by the DBMS)
- But, what should it do if we want to delete $\langle 53666, Jones, Jones@cs, 18, 7.4 \rangle$?

Foreign Key Constraints...

- A foreign key can also refer to the same relation
- Example, add a *partner* field to the relation *Students*
- What is the potential problem?

NULL

- A special value that means:
  - It is not known, or
  - It is not applicable
- The appearance of a NULL does not violate the foreign key IC
- But a NULL is not allowed in a primary key

Foreign Key Constraints in SQL

```
CREATE TABLE Enrolled (
        studid  char(20),
        cid     char(20),
        grade   char(10),
        PRIMARY KEY  (studid, cid),
        FOREIGN KEY  (studid) REFERENCES Students
);
```

Enforcing Integrity Constraints

Operation	Domain	Primary Key	UNIQUE	Foreign Key
Insert	YES	YES	YES	YES
Delete	NO	NO	NO	Special
Modify	YES	YES	YES	Special

- Except for Foreign Key IC, all the others are trivial
- The DBMS should take a sequence of **Referential Integrity Enforcement Steps** to guarantee ICs are not broken

Options to deal with Foreign Key ICs

`NO ACTION`	Delete or updates should be rejected
`CASCADE`	Update or Delete the dependent FK
`ON DELETE SET DEFAULT`	The dependent FKs are set to a default FK
`ON DELETE SET NULL`	The dependent FKs are set to NULL

- Example:

```
CREATE TABLE Enrolled (
        studid  char(20),
        cid     char(20),
        grade   char(10),
        PRIMARY KEY  (studid, cid),
        FOREIGN KEY  (studid) REFERENCES Students
                ON DELETE CASCADE
                ON UPDATE NO ACTION
);
```

Transactions and Constraints

- How do we circumvent ICs when we **know** what we are doing?
- For example, we want to create a **transaction** that:
  1. Adds a course registration first for a new student
  2. Then it adds the student
- The only way to do this is to **DEFER** the constraint verification to the end of the transaction.

Transactions and Constraints...

- SQL allows you to specify when you want to do the IC verification:
  - **IMMEDIATE**: The IC is checked at the moment the statement is executed
  - **DEFERRED**: The IC is checked at the end of the transaction

```
SET CONSTRAINTS ContraintFoo DEFERRED
```

Querying Relational Data

- A **relational database query** is a question we pose on the data
- The **result** is always a **relation** containing at least 1 column and 1 row.
- Example:

```
SELECT * FROM students S WHERE s.age < 18;
```

- Result:

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	8.0
53832	Guldy	guldu@music	12	2.0

More queries

- Retrieve only name and login:

```
SELECT S.name, S.login FROM students S WHERE s.age < 18;
```

- Result:

name	*login*
Madayan	madayan@music
Guldy	guldu@music

Combining Tables

- We need to know names of students who got A and the id of the course

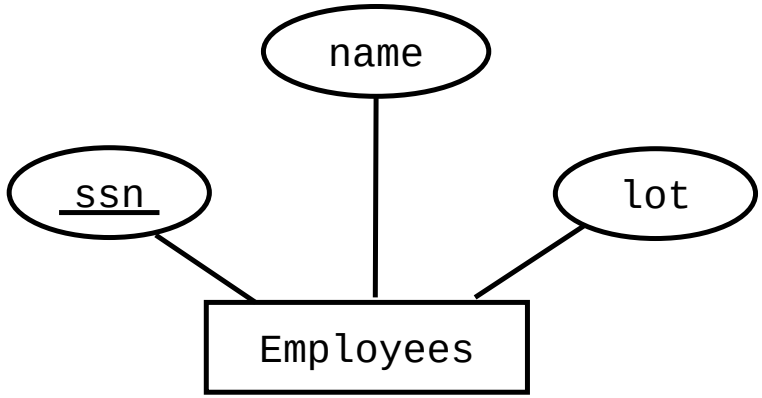
```
SELECT S.Name, E.cid FROM Students S, Enrolled E
    WHERE S.sid = E.studid AND E.grade = 'A';
```

- Result:

name	*cid*
Smith	Topology112

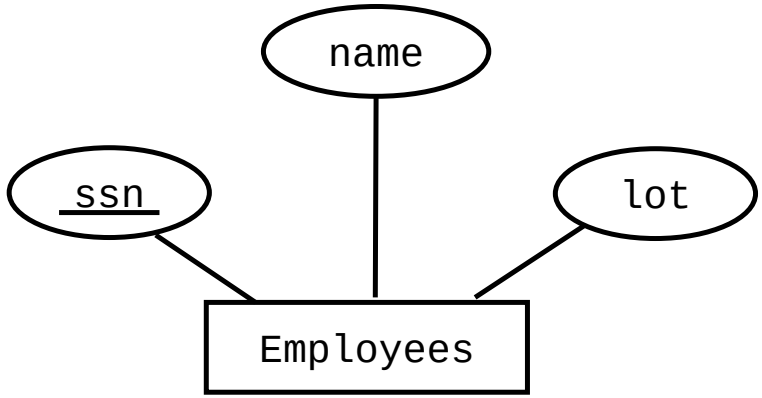
ER to SQL: Mapping Entity Sets to Tables

- It is straightforward:
  - Each attribute becomes a field
  - We also know the primary key of the set
  - We need to determine the domain of each field



ER to SQL: Mapping Entity Sets to Tables

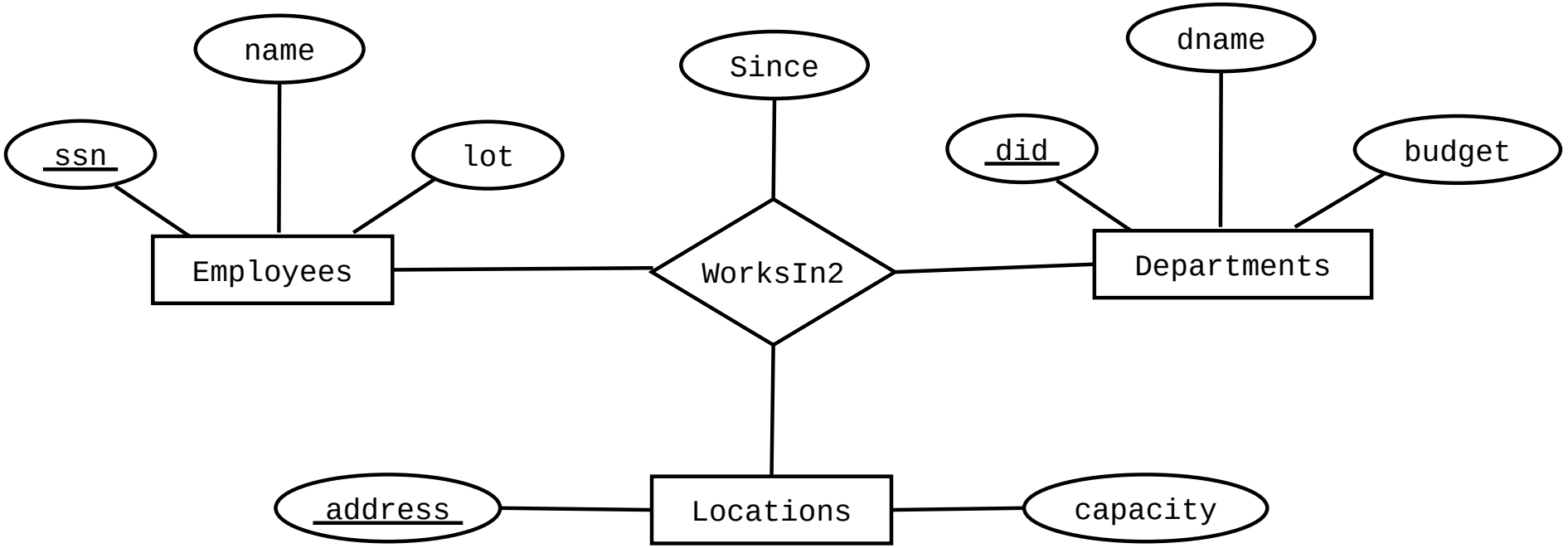
- It is straightforward:
  - Each attribute becomes a field
  - We also know the primary key of the set
  - We need to determine the domain of each field



Relationship Sets

- To map relationships without constraints to a table:
- The fields of the table are:
  - A primary key attributes of each participating entity set, as foreign keys
  - The descriptive attributes of the relationship set
  - The set of non-descriptive attributes is a **superkey**, and becomes a **candidate key**

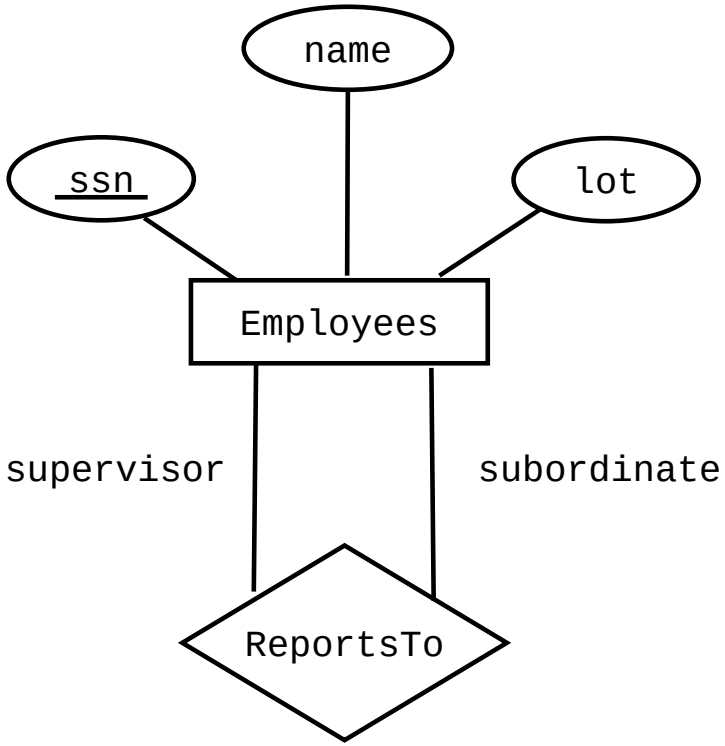
Relationship Sets...



Relationship Sets...

- *address*, *did*, *ssn* cannot take NULL values
- Enforced by making *(address*, *did*, *ssn)* the primary key

Relationship Sets...



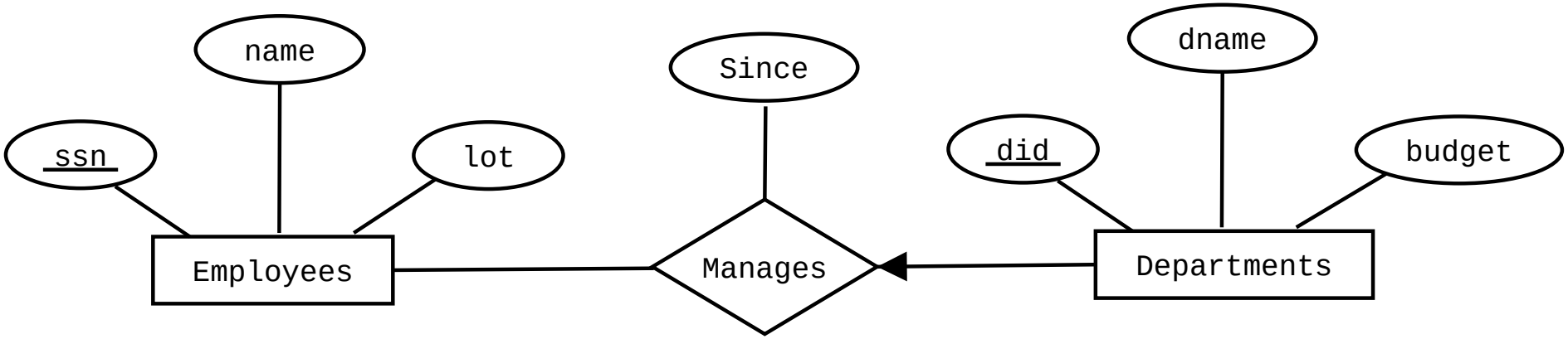
- In this case we want to create meaningful field names

Relationship Sets with Key Constraints

If we have *m* entities related using an *arrow* in a relationship:

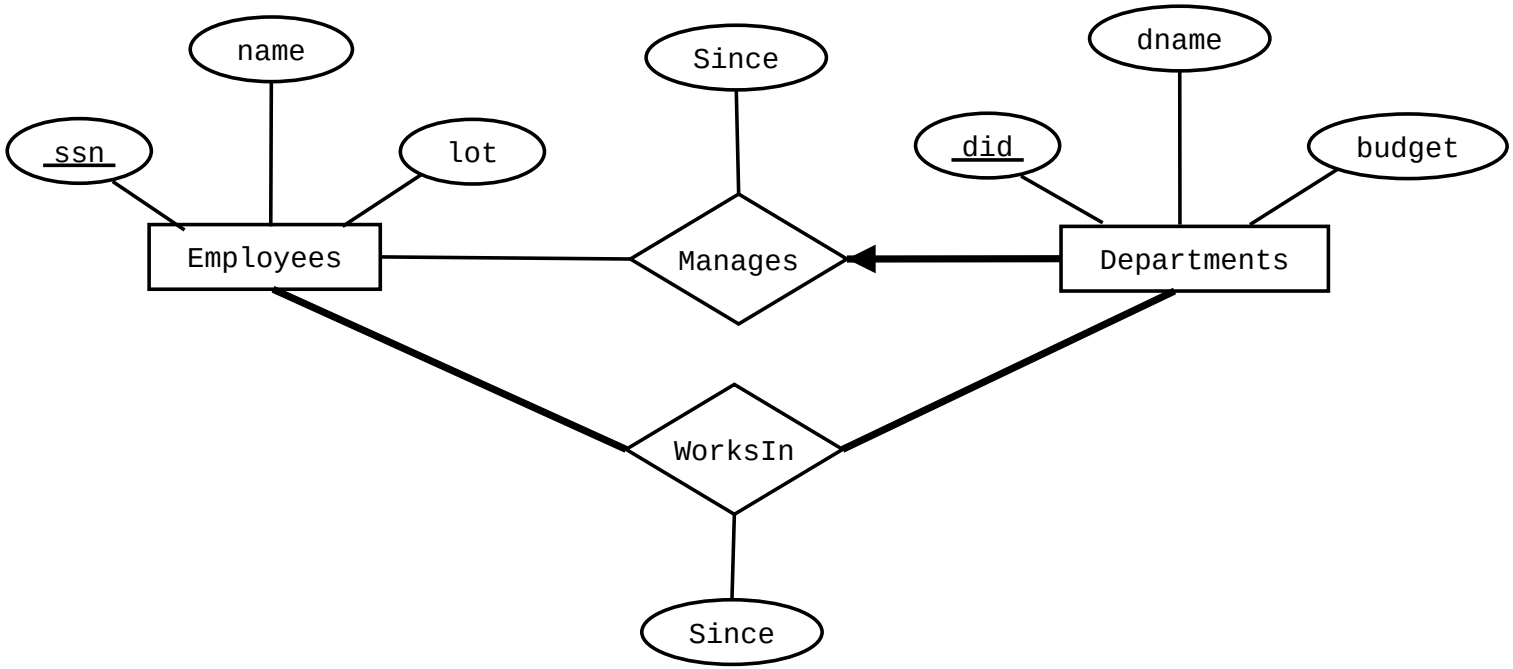
- There are two approaches:
  1. We have *m* candidate keys
  2. One of them should be the primary key
- Add the relationship (fields) to the entity that has the contraint

’Arrow’ relationship



Participation Contraints

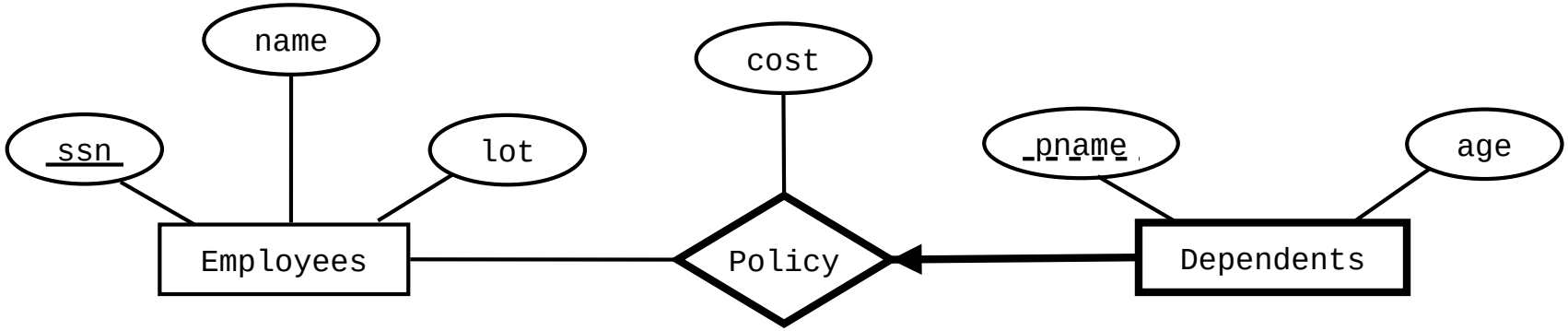
- How do we translate *manages*?



- Add it to the *Department* entity

Weak Entities

- The weak entity has a **partial** key
- When the **owner entity** is **deleted** all the **weak** entities should be deleted
- Solution? Put the relation and the entity in a single table



Further Reading

- Read page 81 (on Constraints that cannot be captured using SQL's CREATE TABLE)

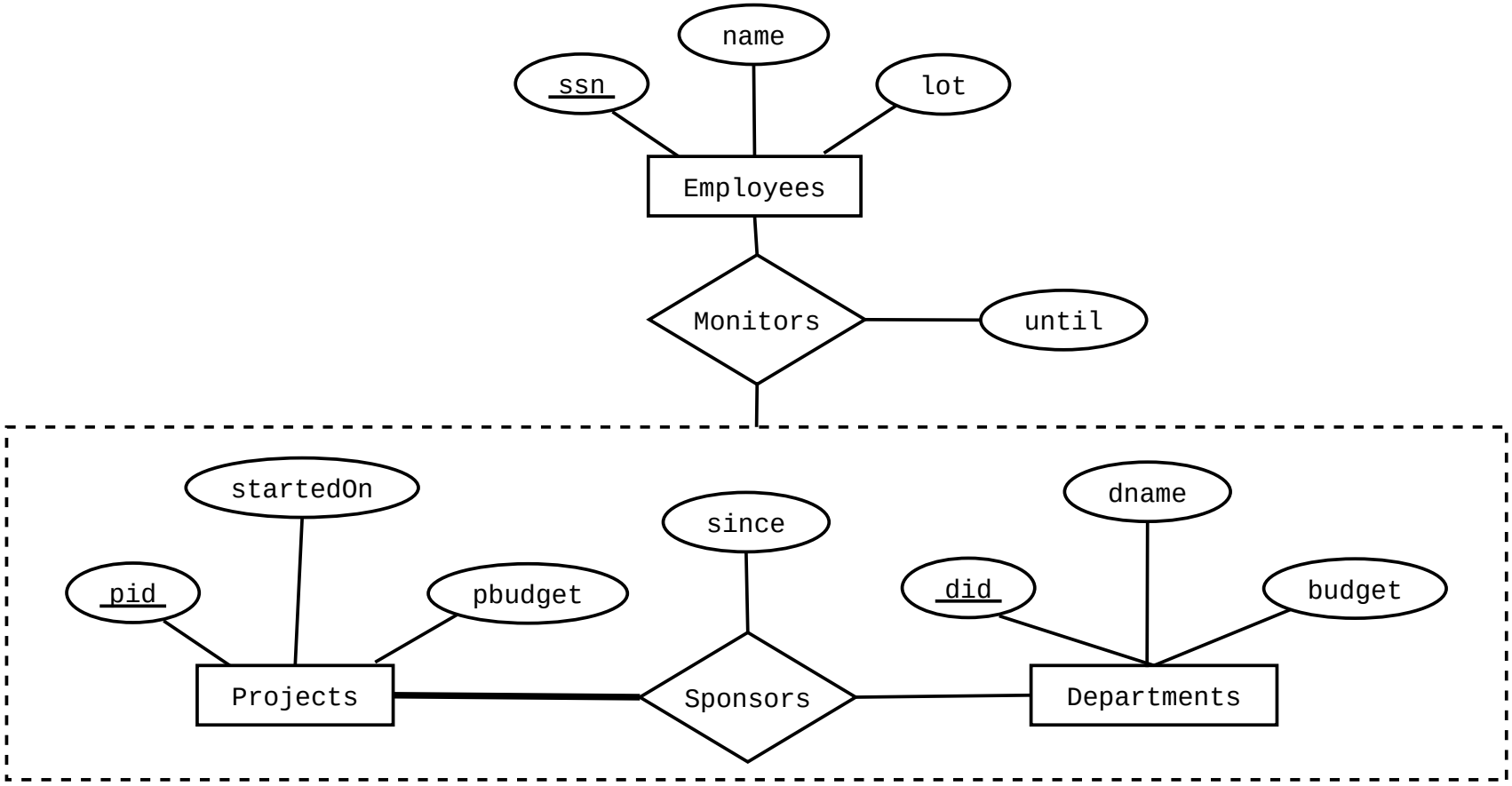
Class Hierarchies

- To translate a class hierarchy there are two general approaches:
  1. Create a different table for each class, but **include the common attributes** only in the ancestor classes.
  2. Create a different table for each class, but **repeat** the common attributes on each class.
  3. *Postgresql* provides a third: define a table as a **subclass** of another table –which is a variation of 2.

Translating Aggregation

- Translate all entity and relationships (except the aggregation) as usual
- Create a table that includes:
  - As primary key the **primary keys** of the participating entity and the participating relation
  - Add any of the attributes of the relation

Aggregation



Views

- A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **view definition**
- Example:

```
CREATE VIEW BStudents (name, sid, course)
        AS SELECT S.name, S.sid, E.cid
        FROM Students S, Enrolled E
        WHERE S.sid = E.studid AND E.grade = 'B'
;
```

- Postgresql does not support '-' in an idenfier (B-Students)
- This table can be used just like a **base table**

Views: logical data independence

- Views allow **logical data independence**: they can be used to define relations in the external schema **that mask changes** in the conceptual schema.
- If the schema of a stored relation is changed we can define a view with the old schema

Views: security

- Views allow to hide information:
- It can be either rows or columns or both

Updates on Views

- What are the implications of updates on views?
- SQL 92 says: you can update views as long as they:
  - Refer to a single base table
  - They use selection and projection only (no aggregates, nor DISTINCT)
  - They are called **updatable views**
- Postgresql does not support **updatable views** (requires the use of rules)

Implications on Updating Views

- Adding rows: the inserted row might **not include the primary key** for the base table
- Deleting/updating rows: **several rows** in the base table could be affected
- Updating rows: the resulting row **might** not be in the view any more!

A more complex update

```
CREATE VIEW ActiveStudents (name, login, club, since)
        AS SELECT S.name, S.login, C.cname, C.jyear
        FROM Students S, Clubs C WHERE S.name = C.mname AND S.gpa > 3;
```

cname	*jyear*	*mname*
Sailing	1996	Dave
Hiking	1997	Smith
Rowing	1998	Smith

(c) Clubs

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	Jones@cs	18	7.4
53668	Smith	smith@ee	18	7.8
53650	Smith	smith@math	19	6.4

(d) Students

name	*login*	*club*	*since*
Dave	dave@cs	Sailing	1996
Smith	smith@ee	Hiking	1997
Smith	smith@ee	Rowing	1998
Smith	smith@math	Hiking	1997
Smith	smith@math	Rowing	1998

(e) ActiveStudents

Updates on views...

- How do we delete: $\langle$ Smith, Smith@ee, Hiking, 1997$\rangle$?
- 3 options:
  - Change **Students**: delete student (bad idea, why?)
  - Change **Clubs**: delete club association $\langle Hiking, 1997, Smith\rangle$ (bad idea, why?)
  - Change **both**
- This example is a **bad** design of a relation and a view
- How can we deal with the deletion of a view? We need to have the **primary keys** of any **base tables** in the view

Destroying Tables

- When we no longer need a table we use `DROP TABLE`
- Sometimes it is not desirable to delete tables:
  - Might be used in **integrity constraints**
  - Might be **base table** of a view
- In that case, we can specify `RESTRICT`, or `CASCADE`
- Postgresql:
  - It does not support RESTRICT nor CASCADE
  - You can delete a table with views
  - When you try to use it you will get an error of the form:

You can also modify a table

- `ALTER TABLE` modifies the structure of a table
- Add a column:

```
ALTER TABLE Students
      ADD COLUMN MaidenName CHAR(10)
;
```

- Every row gets a `NULL` in the new column
- `ALTER TABLE` can be used to:
  - Modifies types of columns
  - Rename names of columns
  - Remove columns (not supported by Postgresql)
  - Add contraints
  - Remove constraints (not supported by Postgresql)