Advanced Database Management Systems

Object and Object-Relational Databases

Coverage in Textbook

- Chapter 20:
  Concepts for Object Database
  - overview of OO concepts
  - mostly familiar to anyone with experience in UML and OOPLs
- Chapter 21:
  Object Database Standards, Languages and Design
  - ODMG standard
  - translation from EER diagrams to OO schemas
- Chapter 22:
  Object-Relational and Extended Relational Systems
  - Object-relational features of SQL-99
  - ORDBMS examples: Informix and Oracle

Motivating Question

- Suppose we need make all data in a Java program persistent.

- What are our options?

Relational Data Model

- Operational definition: 1st Normal Form

- Database consists of relations (tables)
- Relations are composed of tuples (rows)
- Tuples are composed of attributes
- Attributes are constrained by domains
- Domains are primitive (non-structured) data types

Object Persistence Options

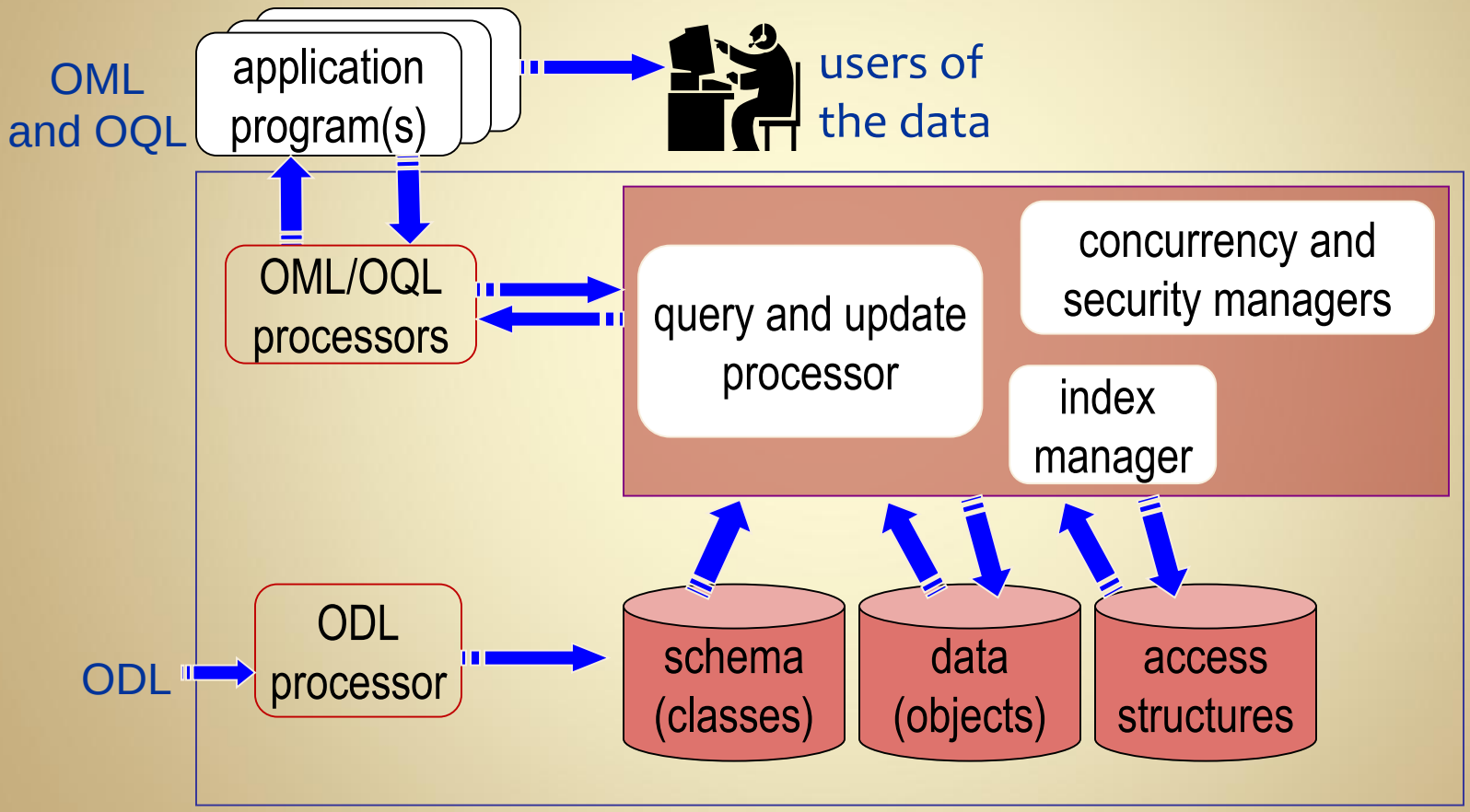
- Serialize the objects and store in files
  - easy in languages that support serialization (Java, Python)
  - problem: no transactions or concurrency control
  - limitation: need to load all data into memory to access any data
- Translate object data into relational data and store in an RDBMS
  - translation process is well understood
  - support for transactions and concurrency control
  - problem: impedance mismatch
- Store objects in an object database

Object Data Models

- Object databases replace the relational data model with an object-based or object-oriented data model

- Database consists of class extents (sets of objects)
- Extents are composed of objects
- Objects are composed of attributes
- Attribute values are constrained by data types
- Data types have arbitrary complexity and structure

OO-DBMS Architecture



Object Databases

- Early OODBMS vendors based their systems on several different object models:
  - Versant, ONTOS: persistent C++ objects
  - O2: object model based on complex value theory
  - Others: persistent Smalltalk objects, Objective-C, etc.
- ODMG: The Object Database Management Group
  - developed a common model for OODBs
  - provides the benefits of standardization in same manner as he standard relational model
  - allows for portability of applications and sharing of objects between systems

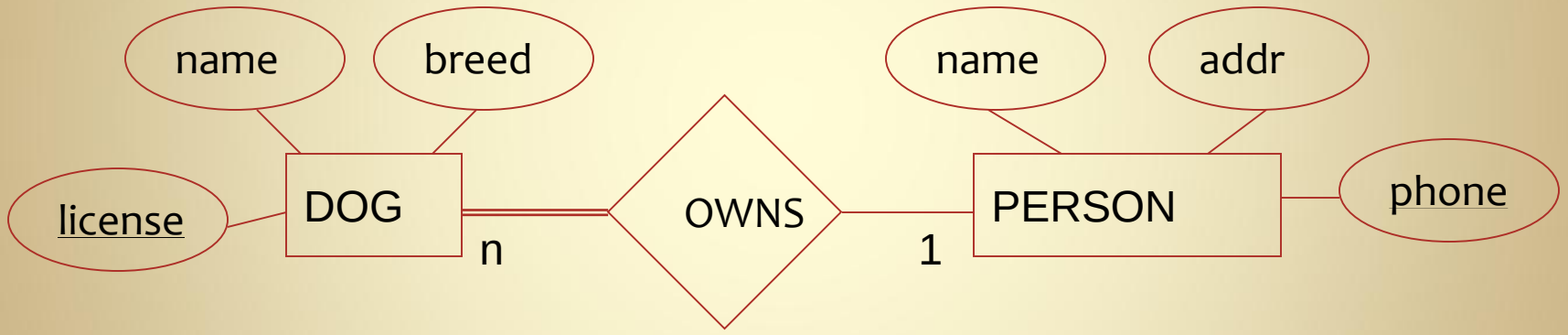
ODMG Standard Components

- Object Model: defines the concepts available for defining an OO schema
  - usual OO things: classes, attributes, methods, inheritance
  - database things: relationships, extents, collections, transactions, DBMS control
- Languages:
  - Object Definition Language: ODL
  - Object Query Language: OQL
  - Object Manipulation Language: OML

ODL: Object Definition

- ODL defines the syntax
  for implementing the object model
  - ODL is the language for defining an object schema
- ODL is actually a family of languages:
  - the ODMG language neutral ODL
  - C++/ODL
  - Java/ODL
  - Smalltalk/ODL
- ODL consists of class declarations

Example Schema: ER



Example Schema: Relational

```
CREATE TABLE PERSON
(
   name   VARCHAR(20) NOT NULL,
   addr   VARCHAR(50) NOT NULL,
   phone  CHAR(10)    NOT NULL;
   CONSTRAINT PERSON_PK PRIMARY KEY(phone)
);
```

Since phone is the primary key,
it becomes the thing that *identifies* a person.

Example Schema: Relational

```
CREATE TABLE DOG
( name          VARCHAR(20) NOT NULL,
  breed         VARCHAR(15) NOT NULL,
  license       VARCHAR(10) NOT NULL;
  owner_phone CHAR(10)      NOT NULL,
  CONSTRAINT  DOG_PK PRIMARY KEY(license),
  CONSTRAINT  DOG_FK FOREIGN KEY
        (owner_phone)REFERENCES PERSON(phone)
);
```

A dog is identified by its license number.

A dog's owner is identified by his/her phone number, since that is a person's primary key.

Relational Instance

PERSON

phone	name	addr
222-7777	Harsha	22 Lake
111-2222	Charlie	16 Pine
333-9999	Regina	801 F

DOG

license	name	breed	owner_phone
001	Snoopy	beagle	111-2222
004	Ace	shepherd	222-7777
003	Rover	poodle	222-7777
012	Spot	mutt	333-9999

relationships are implemented as foreign keys

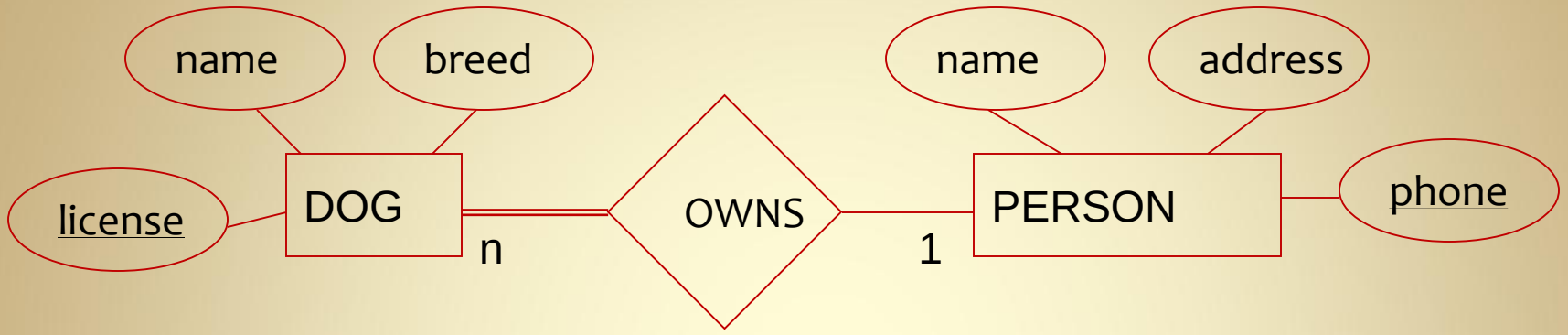
Relational Queries

```
SELECT P.NAME
FROM PERSON AS P, DOG AS D
WHERE D.NAME = "Snoopy"
  AND D.owner_phone = P.phone;
```

```
SELECT D.NAME
FROM PERSON AS P, DOG AS D
WHERE P.NAME = "Harsha"
  AND D.owner_phone = P.phone;
```

relationships are accessed through joins over the foreign key

Object Schema

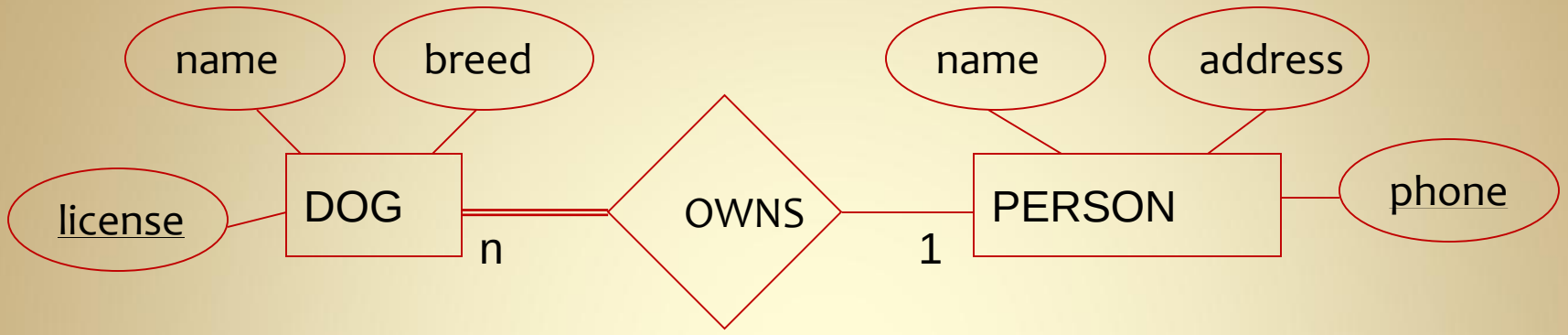



```
class PERSON
(extent people, key phone)
{
  attribute string name;
  attribute string phone;
  attribute string addr;
  relationship set<PET> owns
     inverse PET::owner;
};
```

extent: set of all instances of the class

key: attribute is unique among all instances (not a primary key)

Object Schema




```
class DOG
(extent dogs, key license)
{
  attribute string name;
  attribute string breed;
  attribute string license;
  relationship PERSON owner
     inverse PERSON::owns;
};
```

relationships are *references* to other objects

the inverse relationship implies a consistency constraint

C++ Schema

```
class PERSON : public d_Object {
private:
   d_String name;
   d_String addr;
   d_String phone;
   d_Rel_Set<DOG, "owner"> owns;
};

d_Set<d_Ref<PERSON> > people;
```

all classes inherit from d_Object (database object)

one side of the relationship, defined as a set of relationship references

the class extent is a set of references

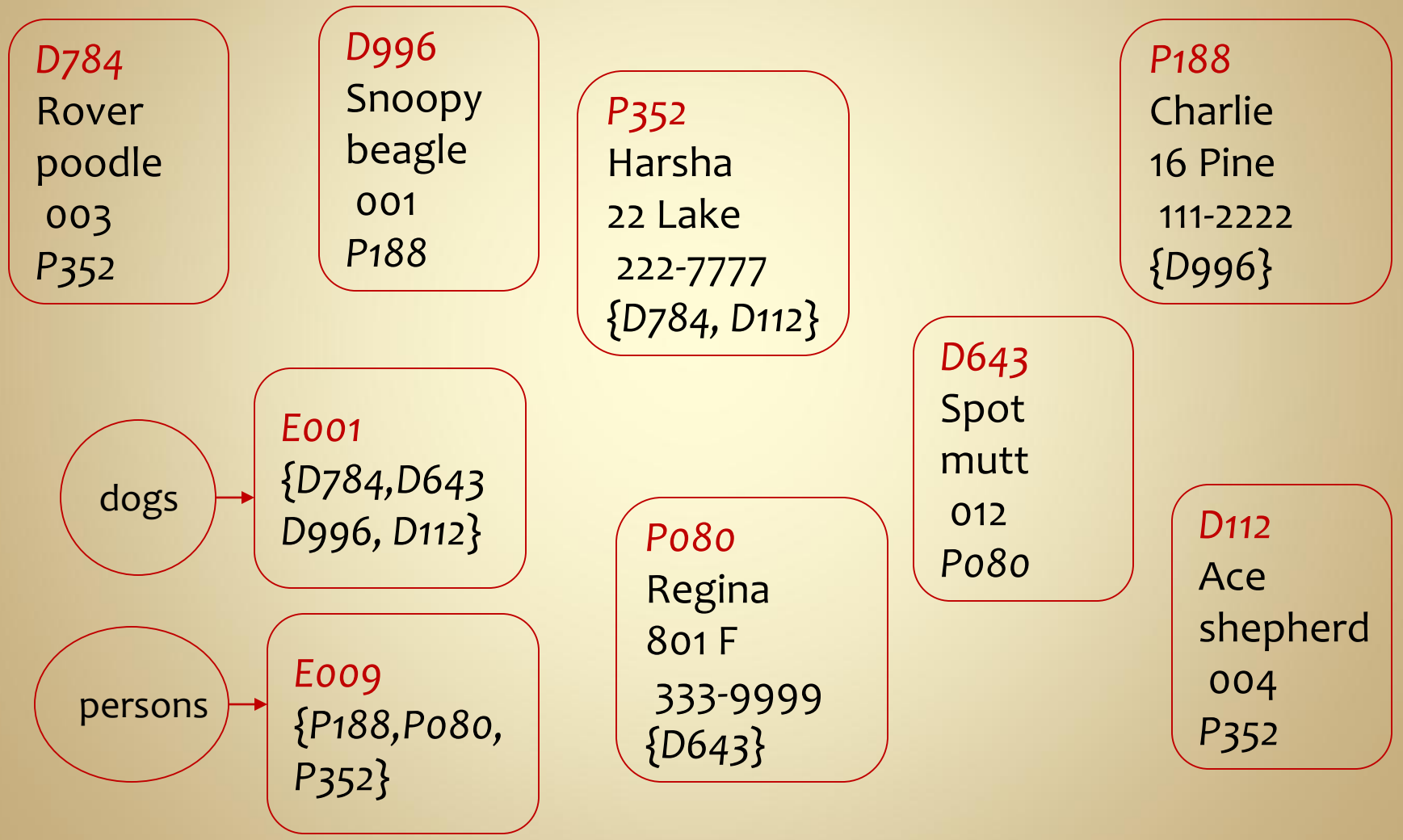
C++ Schema

```
class DOG : public d_Object {
private:
   d_String name;
   d_String breed;
   d_String license;
   d_Ref_Ref<PERSON, "owns"> owner;
};

d_Set<d_Ref<DOG> > dogs;
```

one side of the relationship, defined as a single relationship reference

Object Instances



Extents

extents are sets of
object identifiers (OIDs)

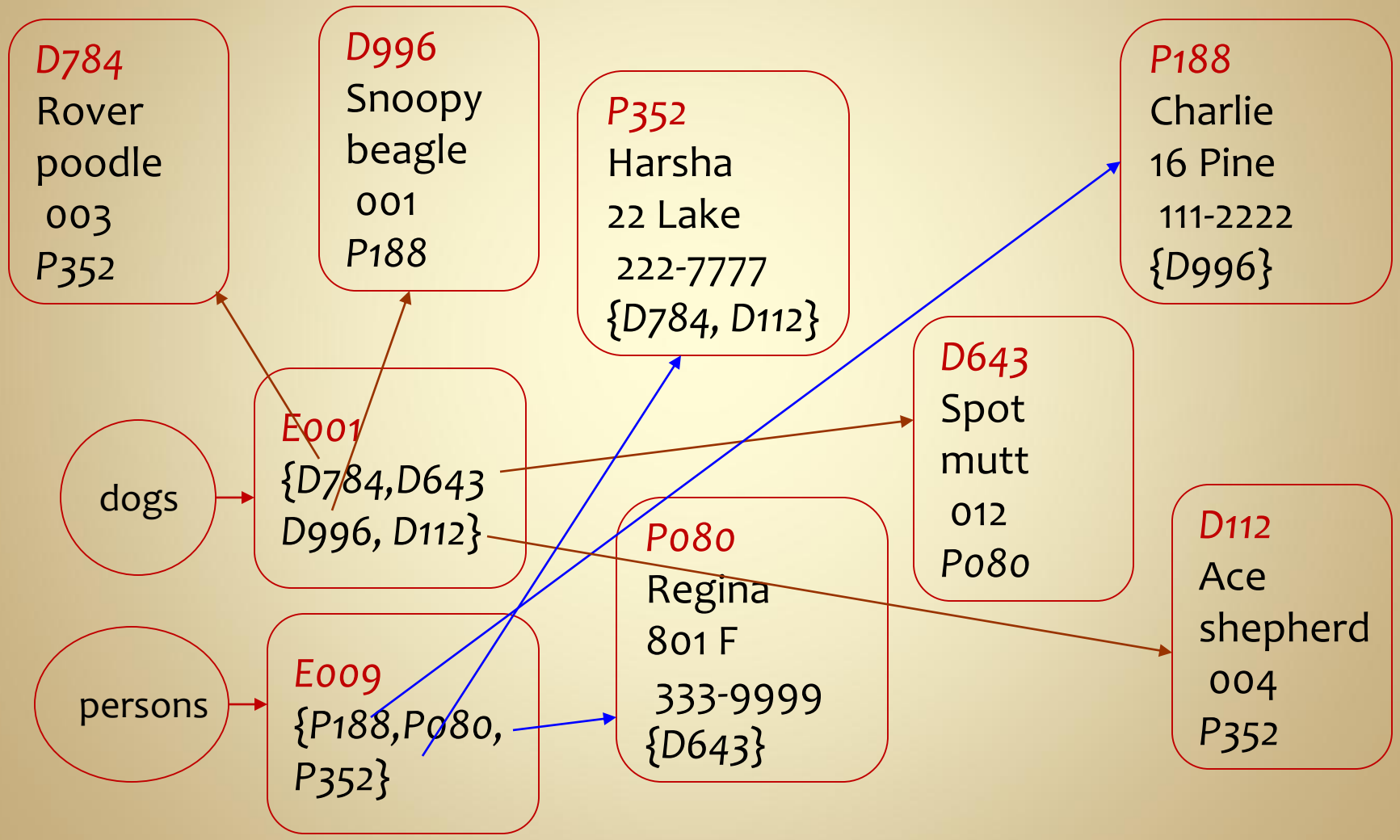
extents have
persistent names

(notation: circles are names,
not pointers or objects)

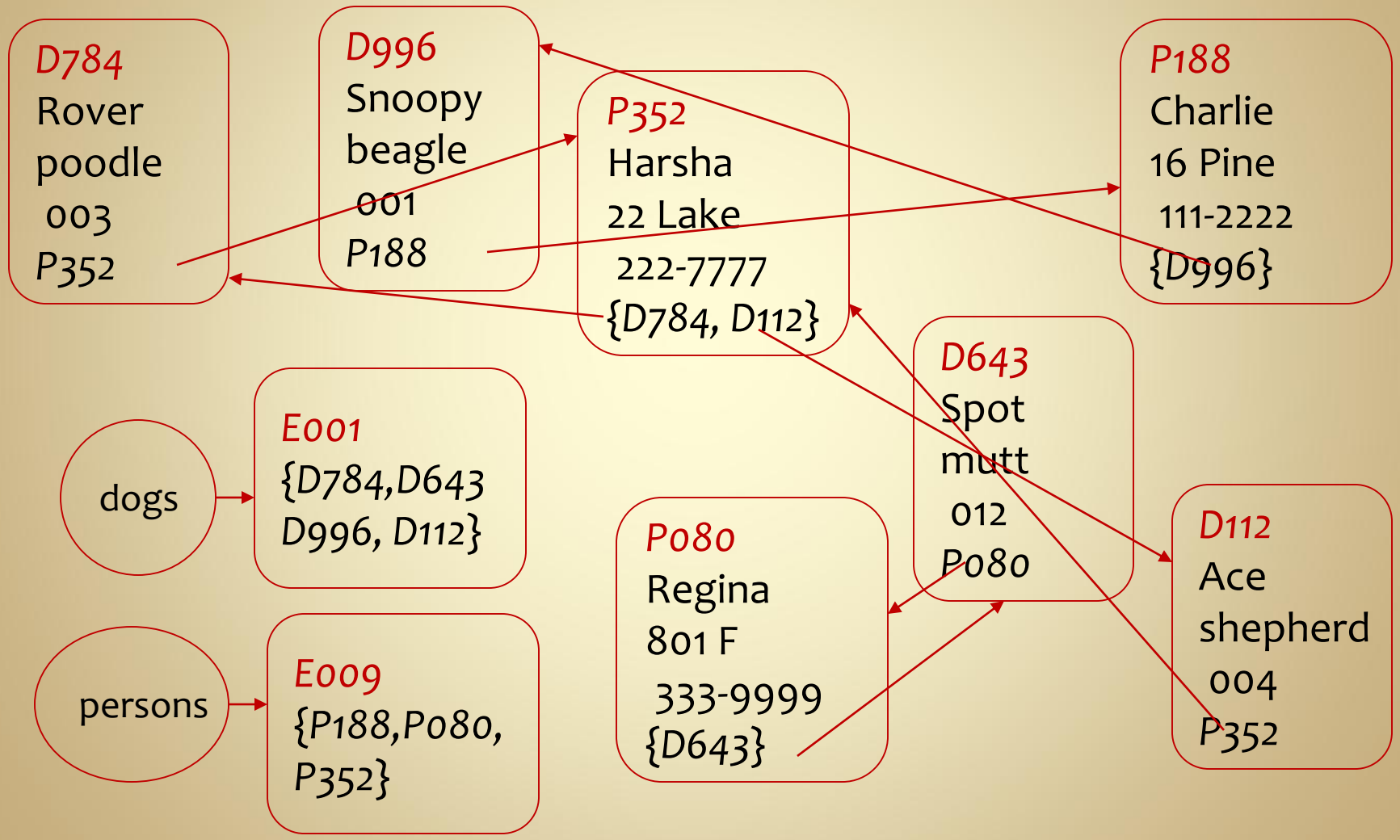
Object Identifiers

- All database objects are assigned unique object identifiers (OIDs)
  - inherited from d_Object
- An OID gives an object an immutable identity, apart from its value
  - In a relational database, the identity of tuples is determined by their value
- An OID identifies an object regardless of its location in memory, on disk or on network

OIDs Are References

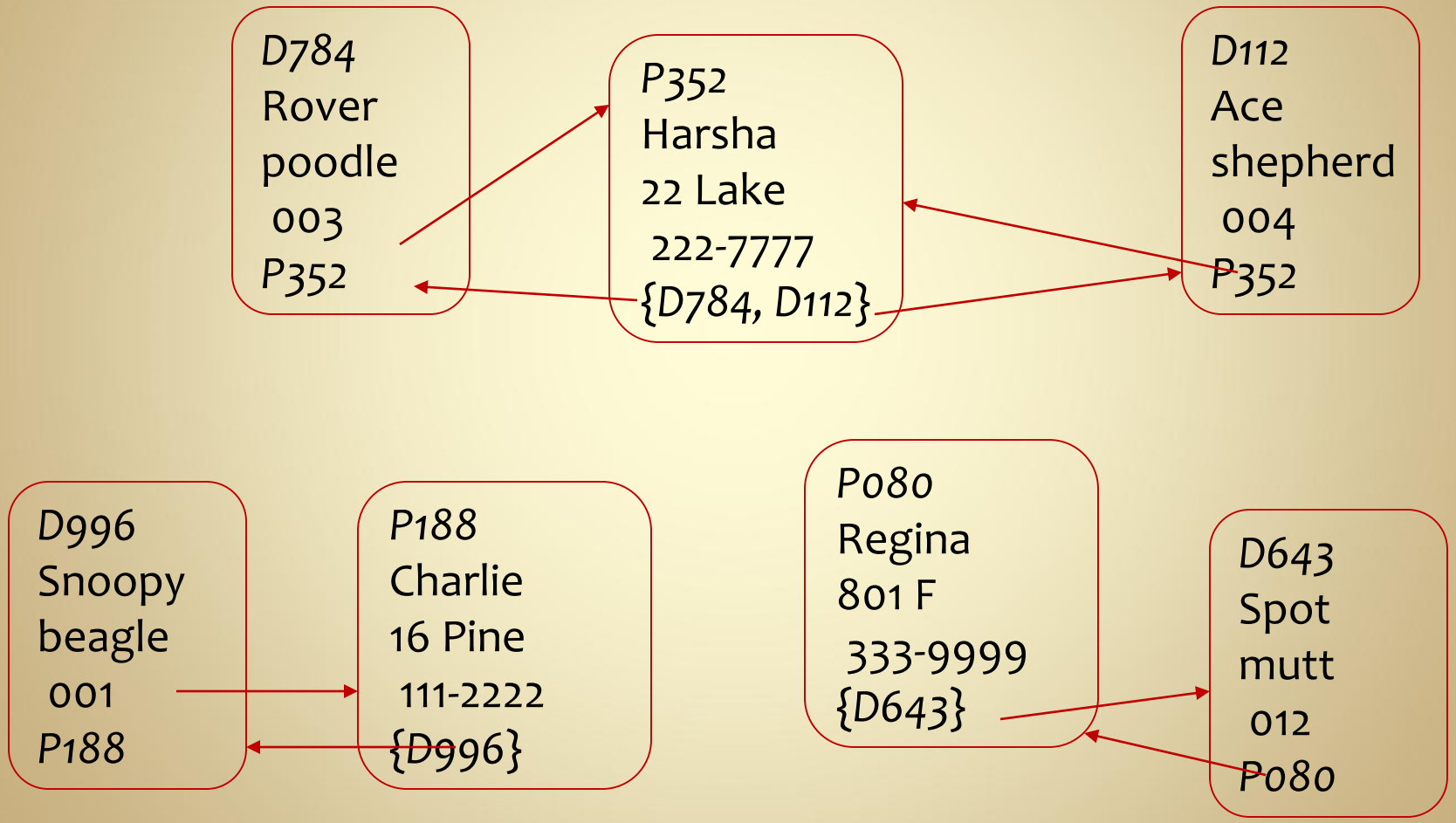


OIDs Are References



relationships are defined by OID values or sets

Relationships are Reciprocal References

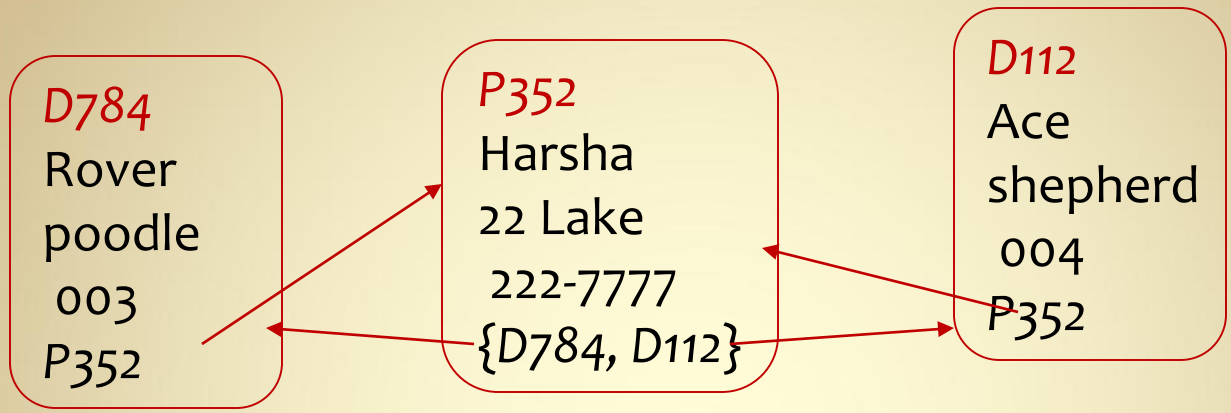


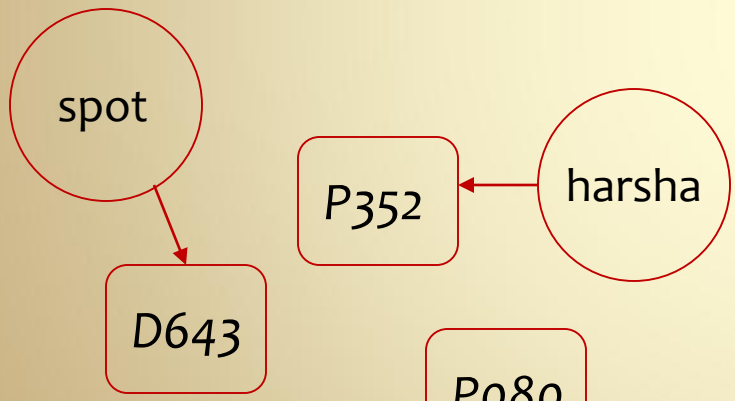
Relationship Maintenance

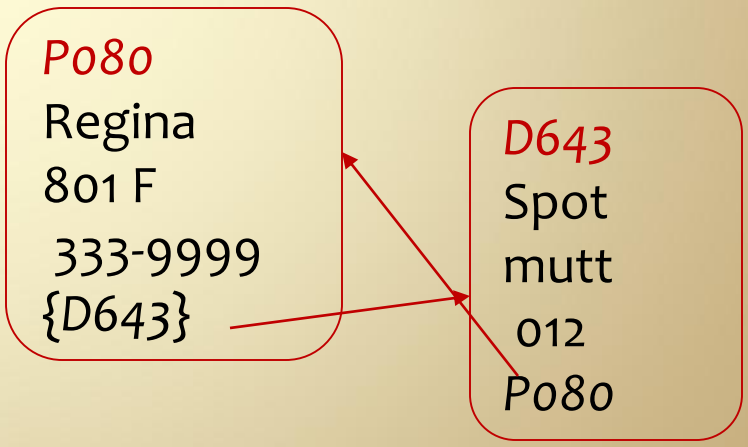
- The DBMS is responsible for maintaining the consistency of relationships
- Suppose Regina sells her dog to Harsha:

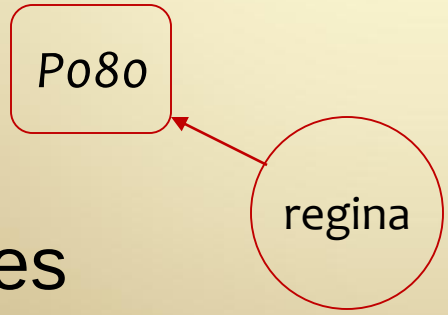
```
d_Ref<Person> regina =
  people.select_element("name='Regina'");
d_Ref<Person> harsha =
  people.select_element("name='Harsha'");
d_Ref<Dog> spot =
  regina->owns.select_element("name='Spot'");
regina->owns.clear();
harsha->owns.insert_element(&spot);
```

Relationship Maintenance



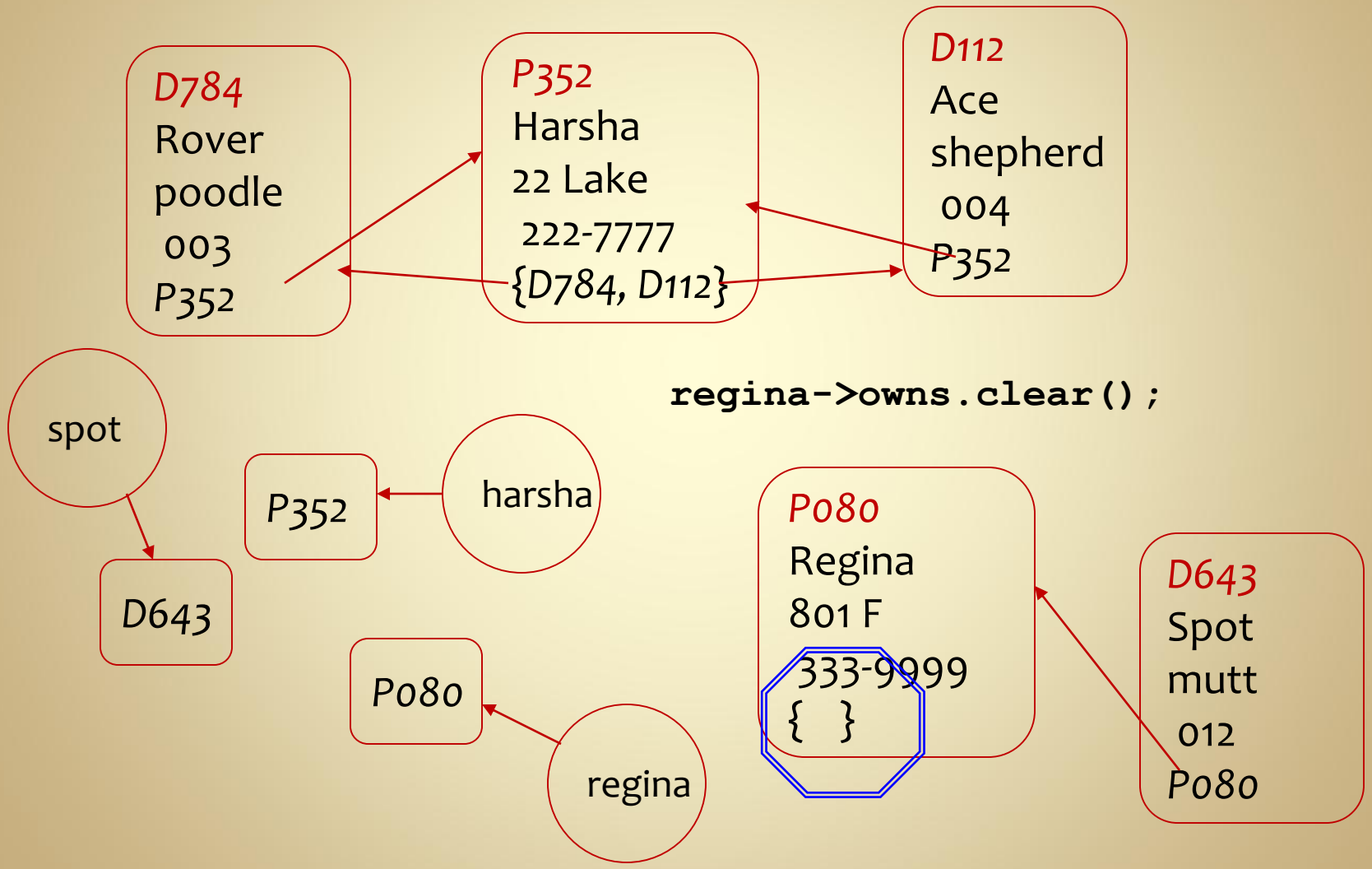




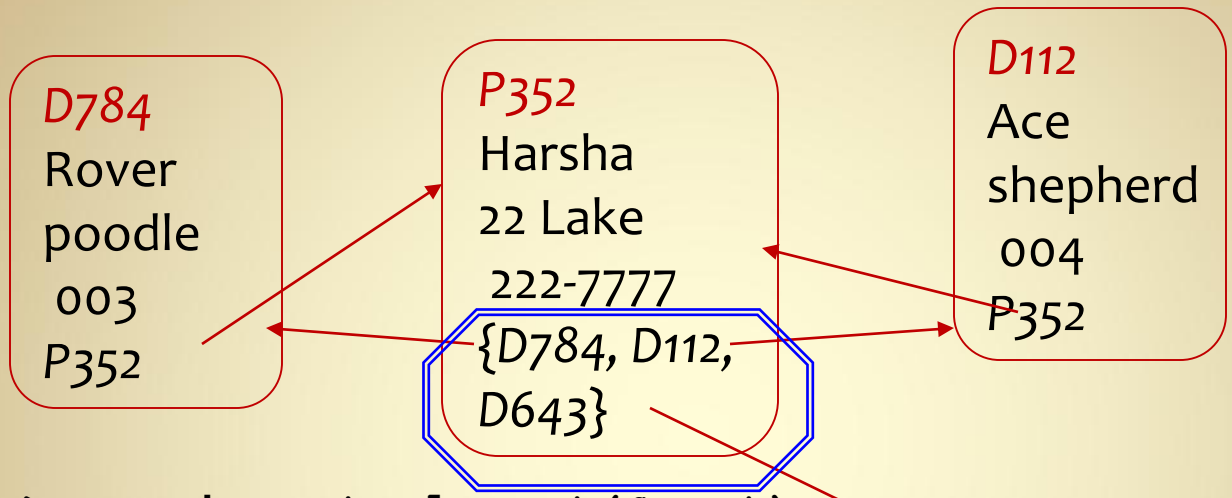


result of the extent queries

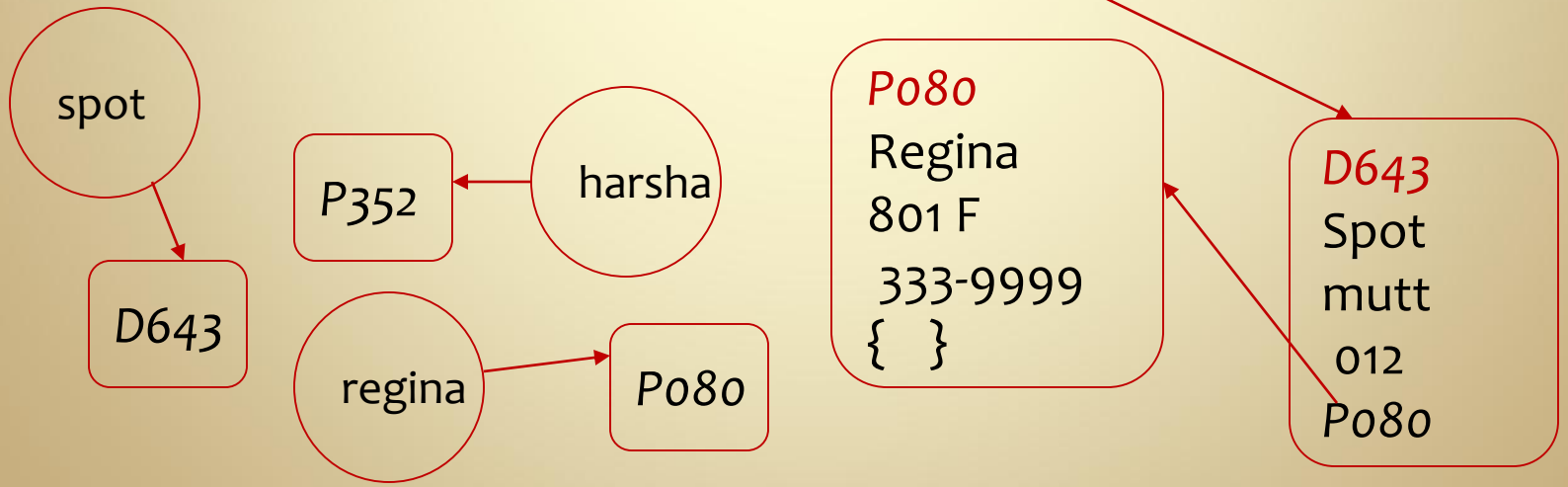
Relationship Maintenance



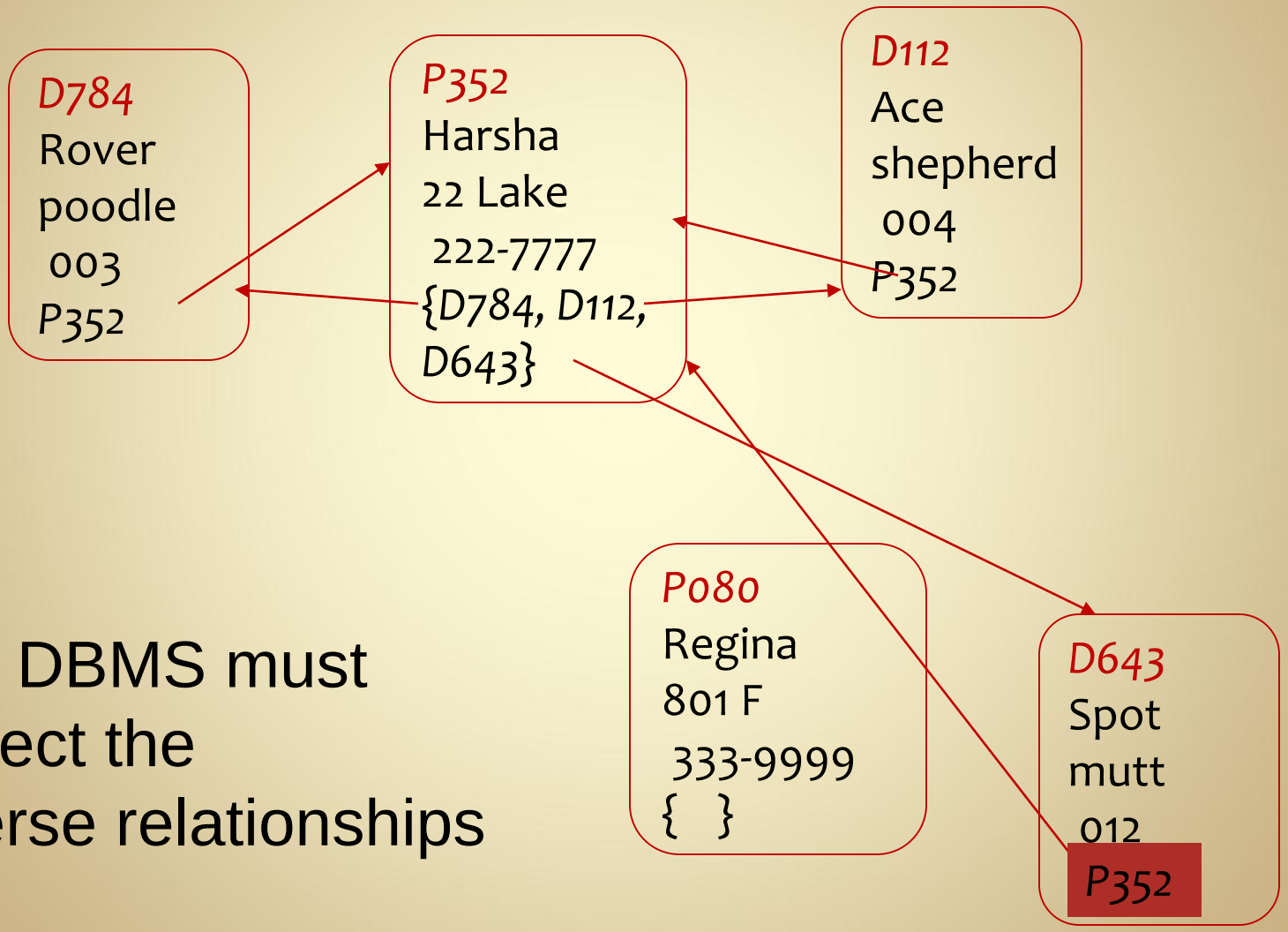
Relationship Maintenance




```
harsha->owns.insert_element(&spot);
```

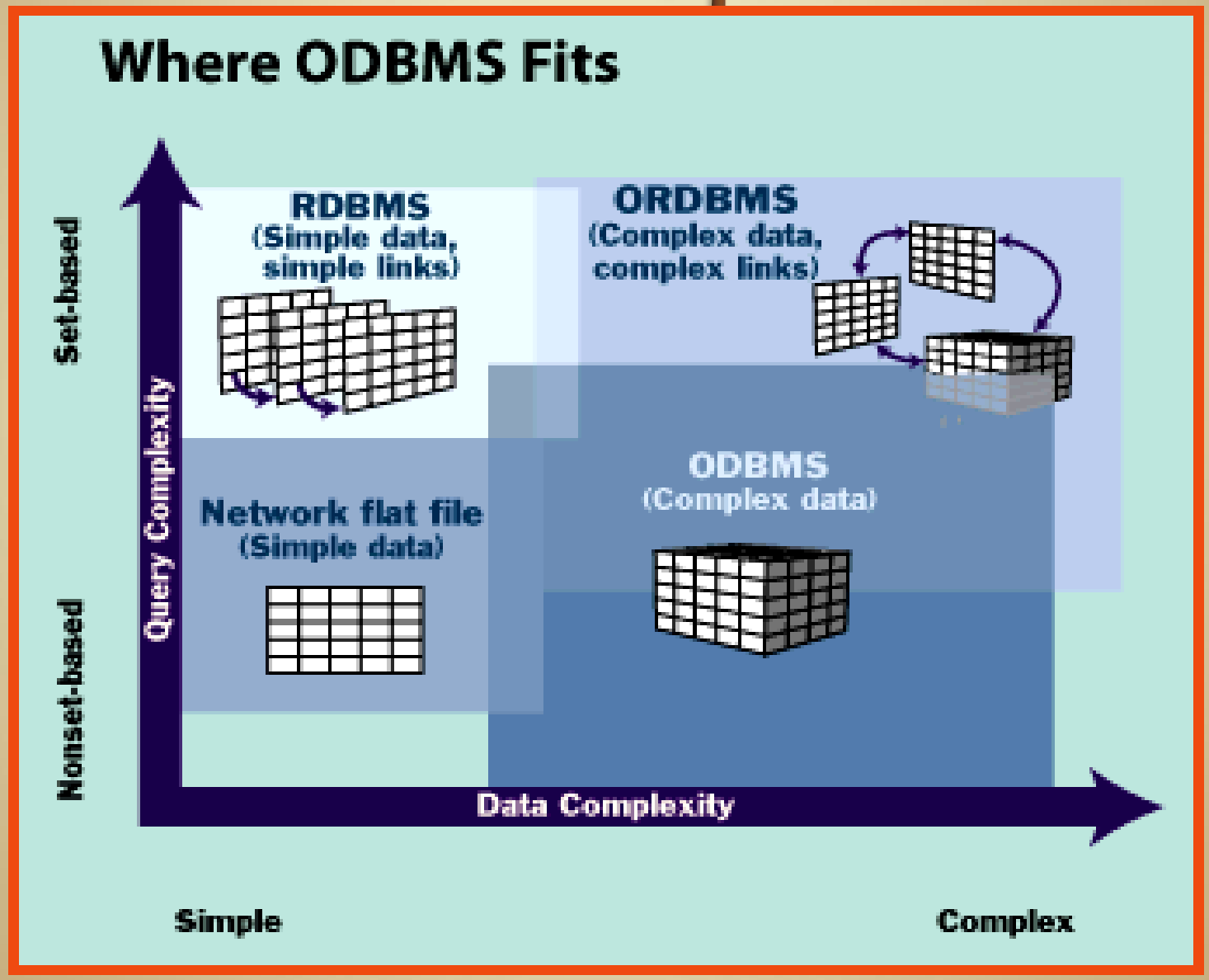


Relationship Maintenance



The DBMS must correct the inverse relationships

Data Model Spectrum



SQL-99: Object-Relational Support

- Type constructors to specify complex objects
  - UDTs → User Defined Types
  - two kinds of UDTs: rows and arrays
- Mechanism to specify object-identity
- Mechanism for encapsulation of operations
- Mechanism to support inheritance

Type Constructor: Row

- Row types consist of flat domains
- Similar to ER composite attribute
- Example:

```
CREATE TYPE Addr_type AS (
     street VARCHAR (45),
     city VARCHAR (25),
     zip CHAR (5));
```

Type Constructor: Array

- An array type is specified for attributes whose values are collections
- Similar to ER multi-valued attribute
- Example:

```
CREATE TYPE Comp_type AS (
  comp_name VARCHAR (2),
  location VARCHAR (20) ARRAY [10]
);
```

OIDs Using References

- A user-defined type can also be used to specify the row types of a table:

```
CREATE TABLE Company OF Comp_type(
    REF IS comp_id SYSTEM GENERATED,
    PRIMARY KEY (comp_name));
```

- Syntax to specify object identifiers:

```
    REF IS <oid_attribute>
           <value_generation_method>
```

- Options:
  - SYSTEM GENERATED
  - or DERIVED

Attributes as References

- A component attribute of one tuple may be a reference:

```
CREATE TYPE Employment_type AS (
  employee REF (Emp_type) SCOPE (Employee),
  company REF (Comp_type) SCOPE (Company));

CREATE TABLE Employment OF Employment_type;
```

- Keyword **SCOPE** specifies the table whose tuples can be referenced by a reference attribute
  - `e.company->comp_name`

Path Expressions

- Path expresssions are used to refer to components of UDTs

- ```
  SELECT E.Employee->Name
  FROM Employment AS E
  WHERE E.Company->Comp_name = 'ABCXYZ';
  ```
  ```

Encapsulation of Operations

- A construct similar to the class definition
- Users can create a named user-defined type with its own methods in addition to attributes:

```
CREATE TYPE Addr_type AS (
        street VARCHAR (45),
        city VARCHAR (25),
        zip CHAR (5)
)
METHOD apt_no ( ) RETURNS CHAR(8);
```

User Defined Methods

- Code for methods is supplied externally
  - implemented in some general purpose programming language

```
METHOD
CREATE FUNCTION apt_no() RETURNS CHAR(8) FOR Addr_type
AS EXTERNAL NAME 'x/y/aptno.class' LANGUAGE 'java';
```

Inheritance in SQL

- **Inheritance** is specified via the **UNDER** keyword

- Example

```
CREATE TYPE Manager_type
UNDER Emp_type
AS (dept_managed CHAR (20));
```

- `Manager_type` **inherits** all features of `Emp_type` and has an additional attribute called `dept_managed`