COMP163: Database Management Systems

Object-Oriented and Object-Relational Databases Chapter 11

Relational Data Model

- Pros:
  - simple: one basic structure
  - well-founded – solid theory
  - efficient and optimizable
- Cons:
  - simple: no complex structures
  - does not fit data structures of modern programming paradigms

Objects + DBs: approaches

- Translation
  - objects in program ←→ relations in DB
- Augment Relational DB
  - add support for complex type within the RDM
  - object-relational databases
- Full OO
  - build database around OO data model
  - object-oriented databases

Translation: Objects ←→ Relations

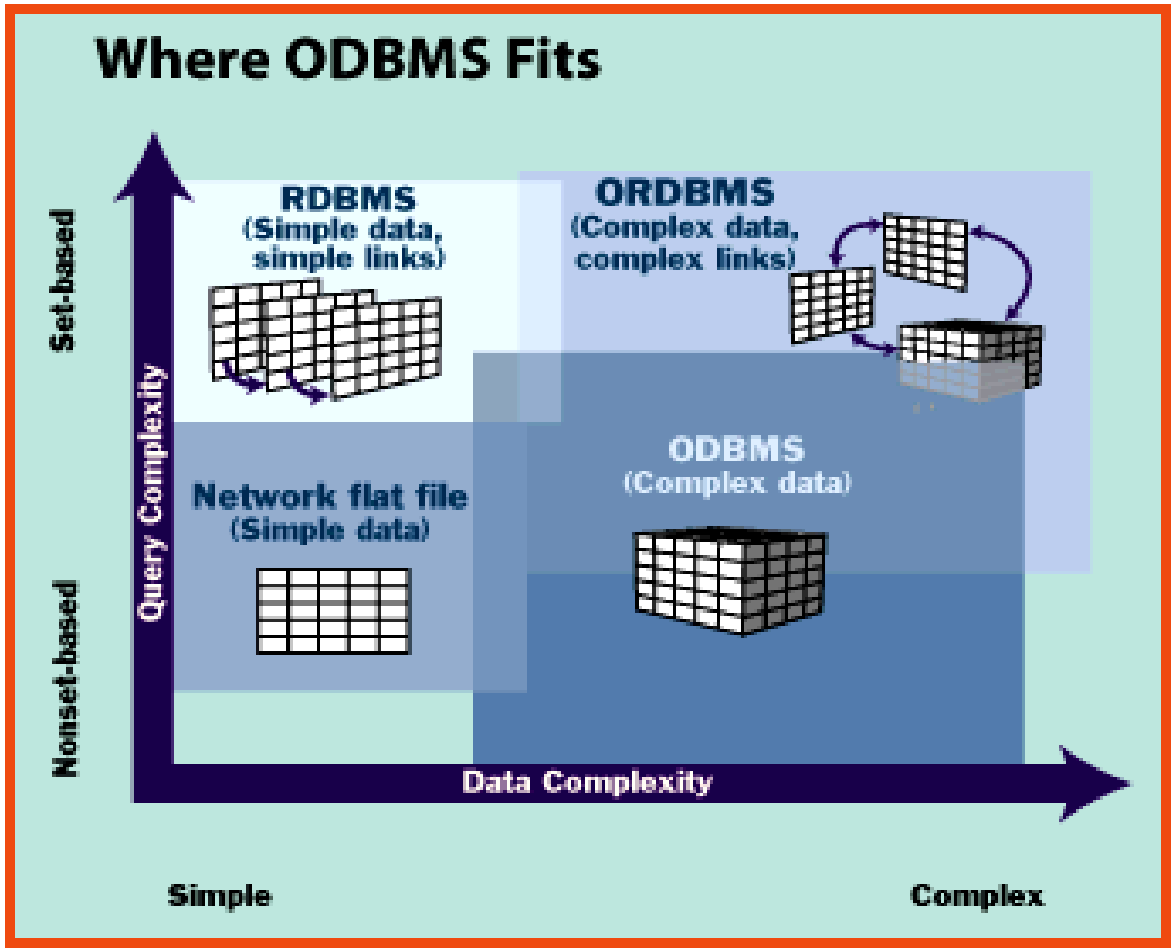
- Ad-hoc:
  - API provides SQL interface to database
  - programmer responsible for mapping relations into objects and back
  - common in data driven web applications (PHP)
- Systematic:
  - defined process for mapping
  - similar to serialization/marshalling/pickling

Mapping: relational – OO

relational	object-oriented
relations	classes
rows/tuples	objects
columns/attributes	attributes
primary keys	object identifiers
foreign keys / join attributes	pointers/references
	complex structures
	inheritance

Algorithms/implementations for this mapping exist.
Example: JDO (Java Data Objects)
gives automatic persistence to Java objects

Data Model Spectrum



Object-Relational DBs

- Allow User Defined Types (UDT)
- Put UDT instances in tables
  - breaks 0NF
- Allow method calls on UDT instances in SQL commands
- Add direct references between UDT instances

Object-oriented DBs

- DBMS built on OO data model
- Direct persistence of objects in application
- Issues:
  - reachability of objects
  - object identifiers
  - relationshipd
  - query optimization
  - joins vs. pointer chasing

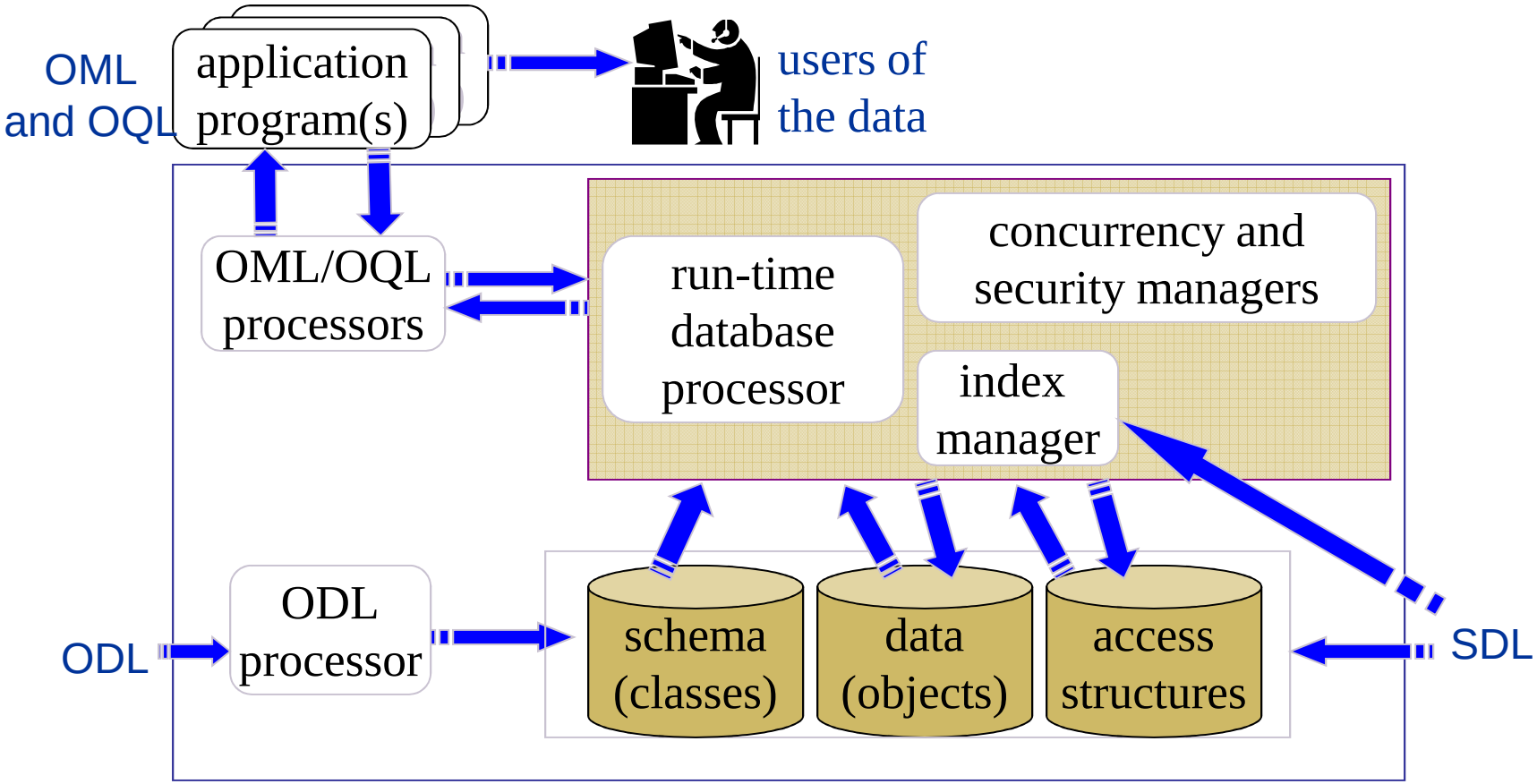
ODMG: The Object Database Management Group

- Early OODBMS vendors based their systems on slightly different object models:
  - Versant, ONTOS: persistent C++ objects
  - O2: object model based on complex value theory
  - Others: persistent Smalltalk objects, Objective-C, etc.
- ODMG standardization developed a common model for OODBs
  - similar to the standard relational model.
  - This allows for portability of applications and sharing of objects between systems.

Components of the ODMG Standard

- 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

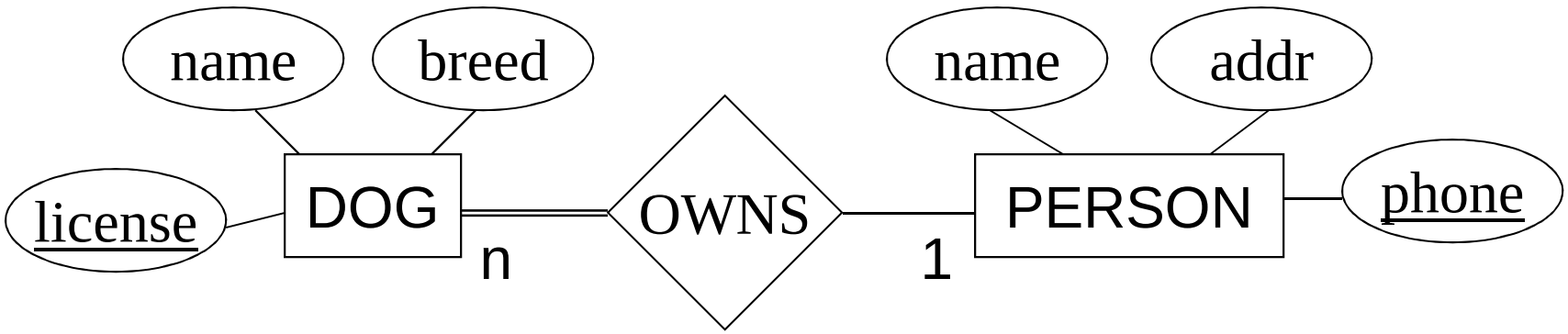
OO-DBMS Architecture



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,
  PRIMARY KEY(license),
  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	shepard	222-7777
003	Rover	poodle	222-7777
012	Spot	mutt	333-9999

relationships are implemented as foreign keys

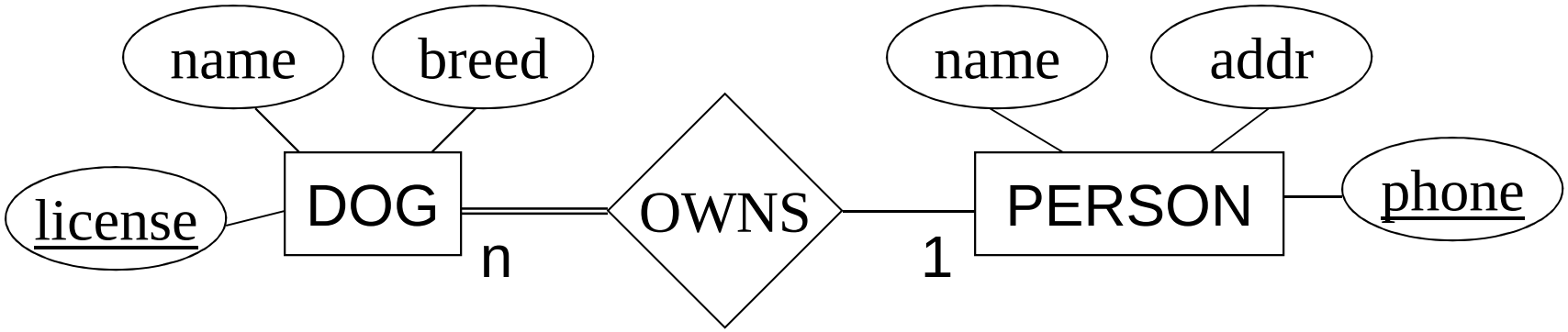
Relational Queries

```
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

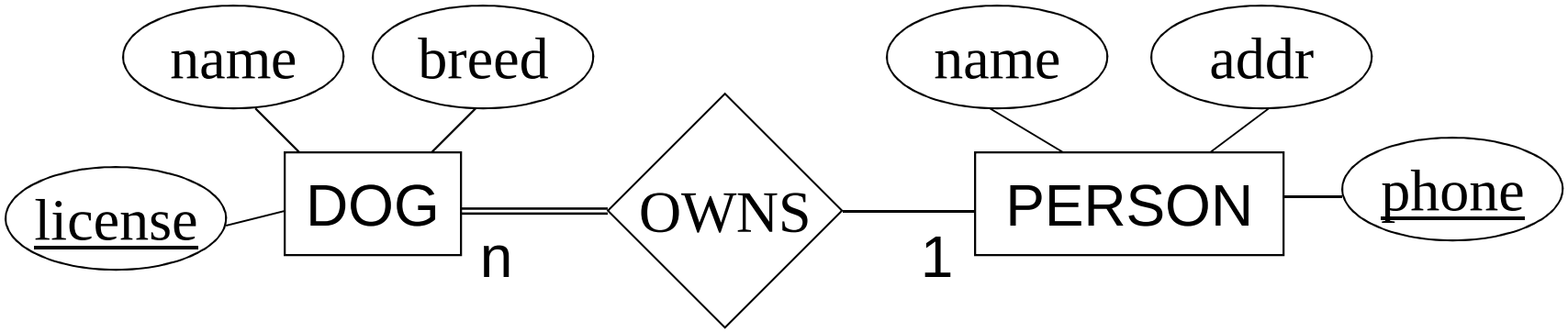



```
class PERSON
(extent people, key phone)
{
  attribute string name;
  attribute string phone;
  attribute string addr;
  relationship set<PET> owns
     inverse PET::owner;
};
```

extent: set of all instances

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)

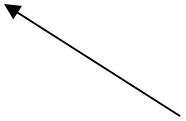
one side of the relationship, defined as a set of relationship references

the class extent is a set of references

C++ Schema

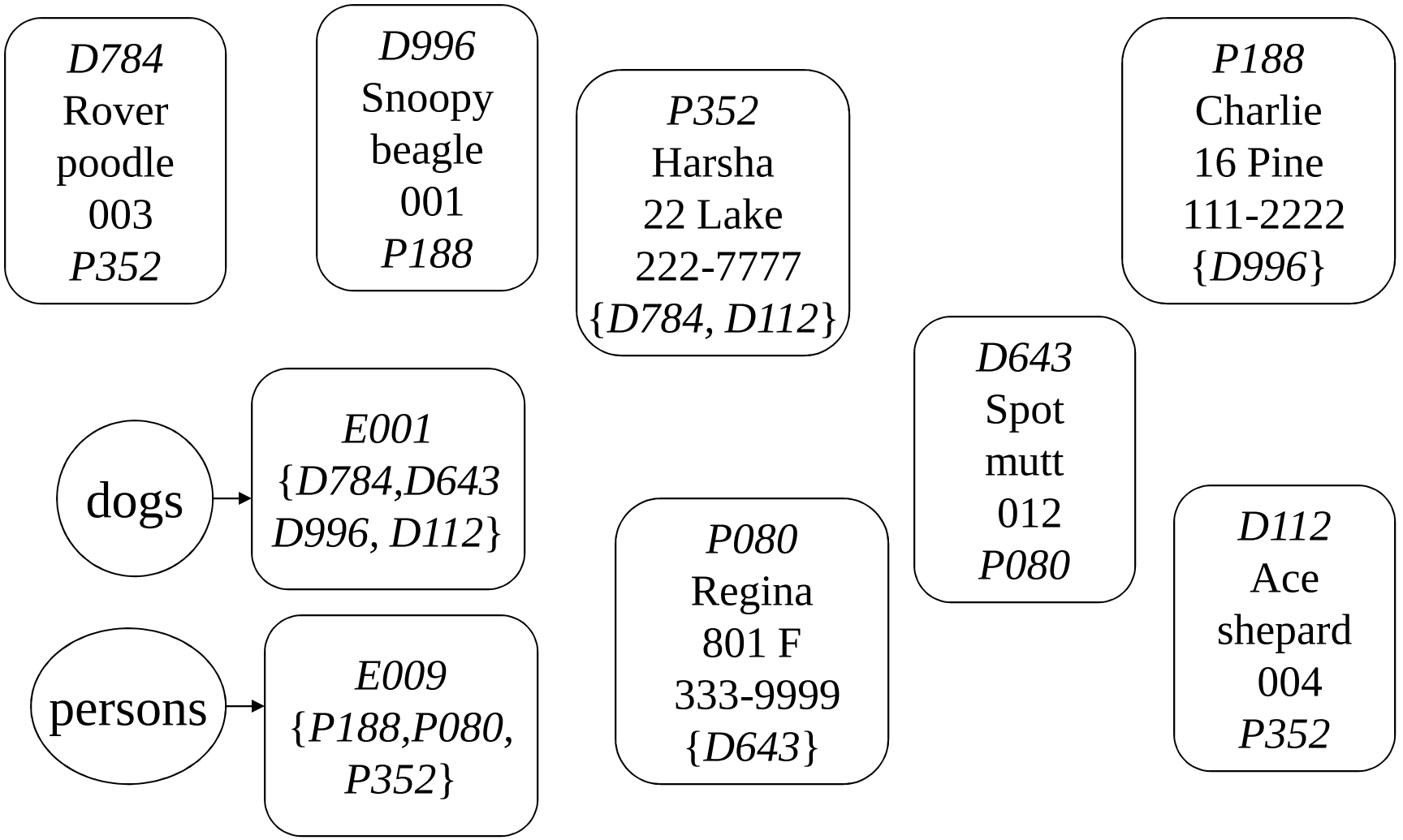
```
class DOG : public d_Object {
private:
  d_String name;
  d_String breed;
  d_String license;
  d_Ref_Ref<PERSON, "owns"> owner;
};

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

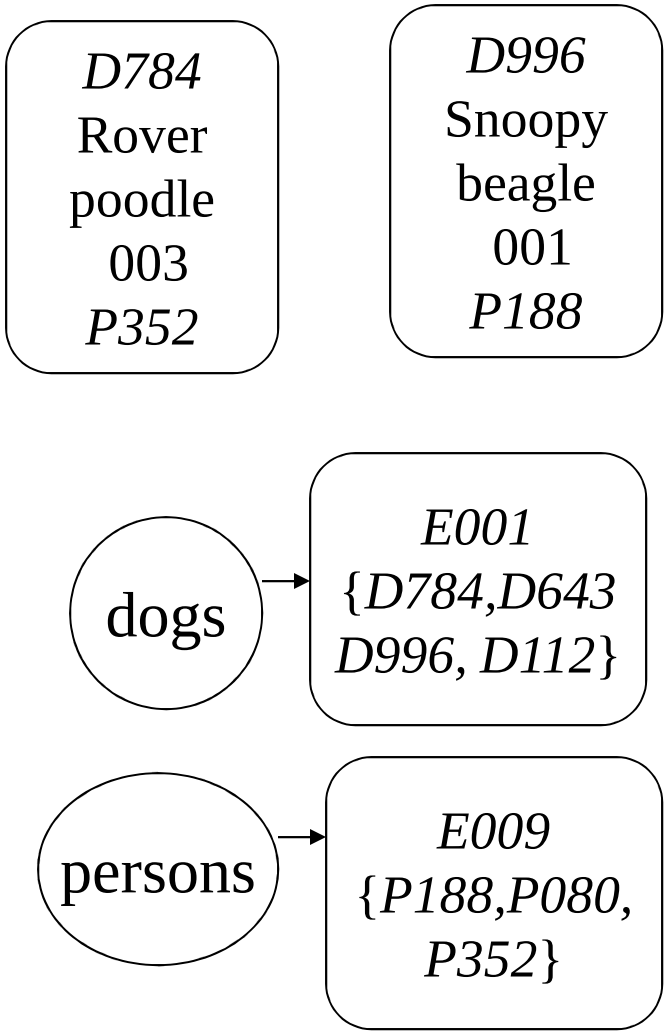


other side of the relationship, defined as a single relationship reference

Object Instances



Object Instances



extents have
persistent names

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

extents are sets of
object identifiers (OIDS)

Object Identifiers

- Questions:
  - How are objects identified in C++ or Java programs?
  - Will this work in a database context?

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

Class Extents

- An extent is simply a set of references to all instances of a class
  - For an OODB, this can be a set of OIDs
- Implementation:
  - hack into all constructors and destructors
- Value:
  - Extent replaces the relation as the basic storage structure
  - Ensures reachability of all objects

Basic Implementation (C++)

```
typedef long OID;

class D_Object
{
public:
  D_Object()              { setOID(); addToExtent(); }
  virtual ~D_Object()     { removeFromExtent(); }

protected:
  static set<OID> extent;

private:
  OID oid;

  static OID next_oid;

  void setOID()           { oid = next_oid++; }
  void addToExtent()      { extent.insert(oid); }
  void removeFromExtent() { extent.erase(oid); }
};
```

Basic Implementation (C++)

```
typedef long OID;

class D_Object
{
public:
  D_Object()              {
  virtual ~D_Object()     {

protected:
  static set<OID> extent;

private:
  OID oid;

  static OID next_oid;

  void setOID()           {
  void addToExtent()      {
  void removeFromExtent() {
};
```

Additional practical issues:

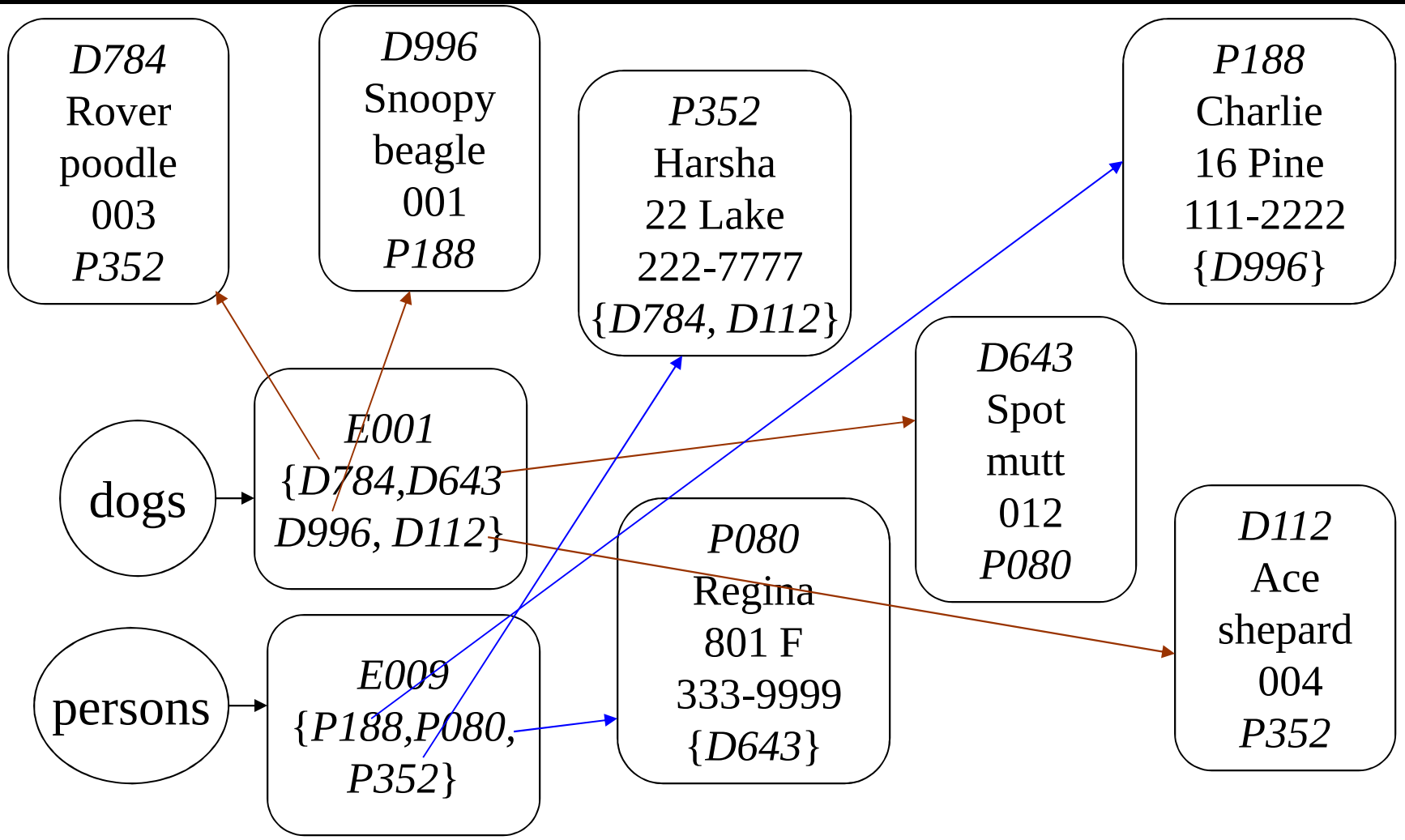
More complex OIDs require an OID generator

Need ability to map OIDs to object pointers/references

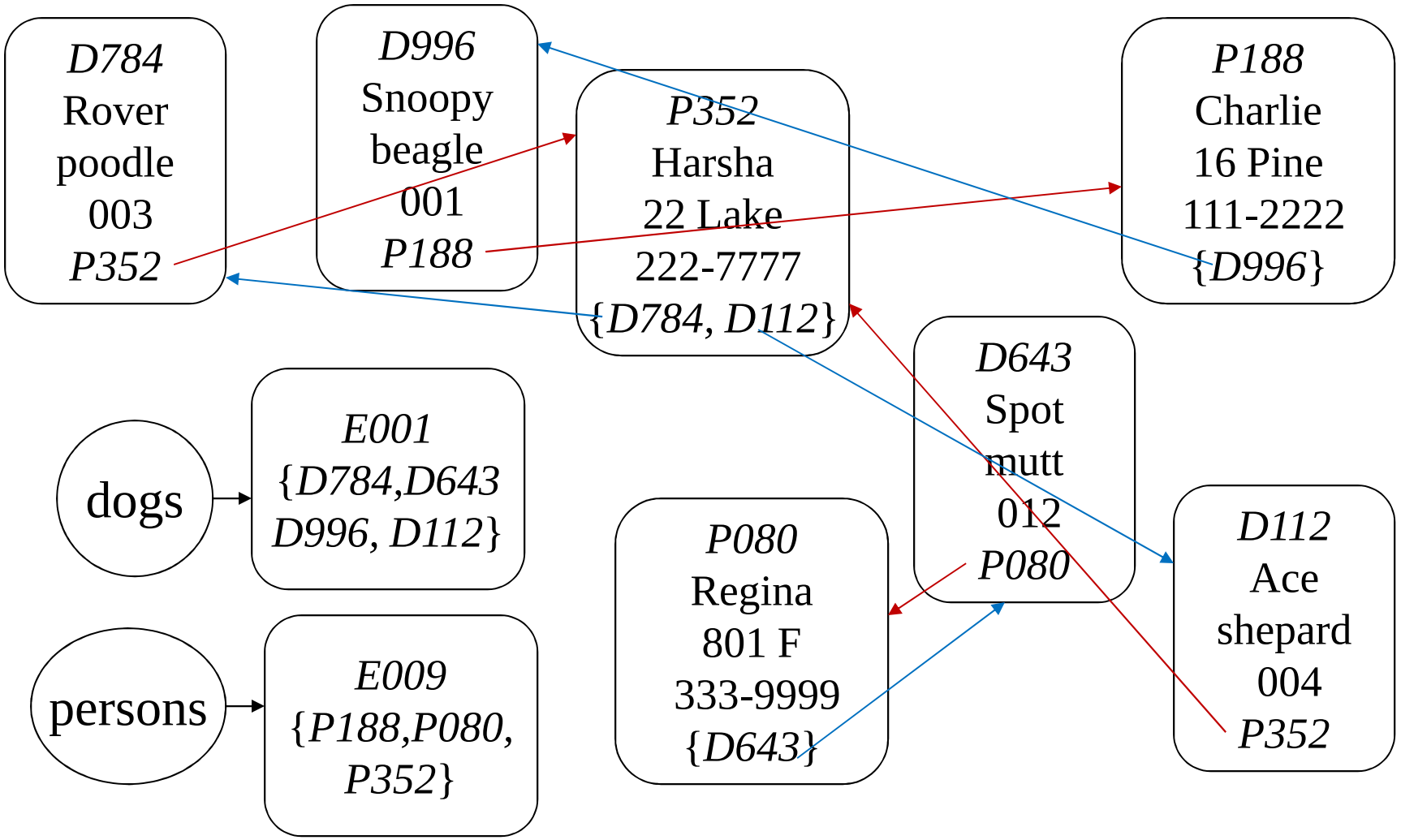
Need separate extents for each class

(This are all easily solvable)

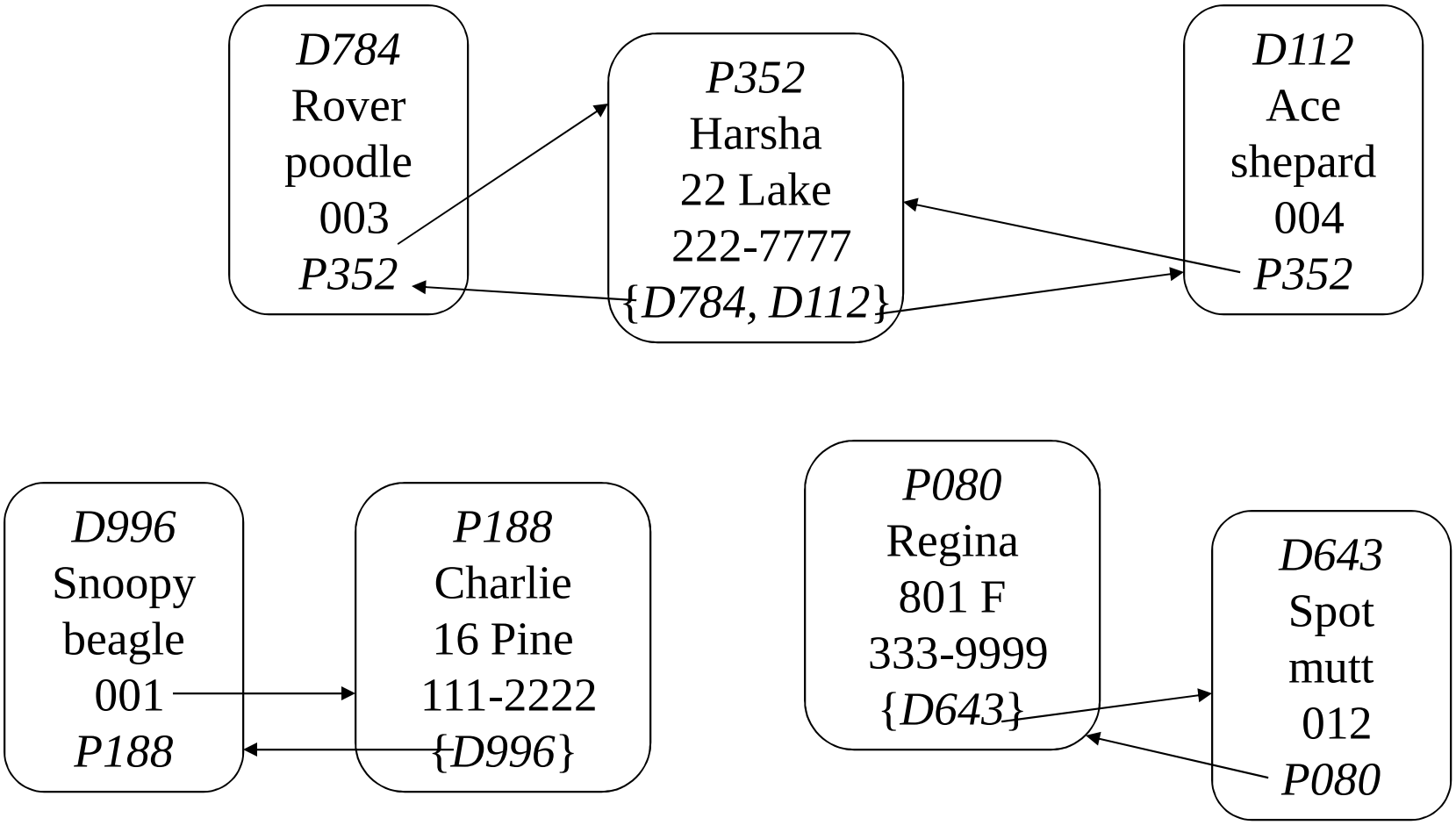
OIDs Are References



OIDs Are References



Relationships are Reciprocal References

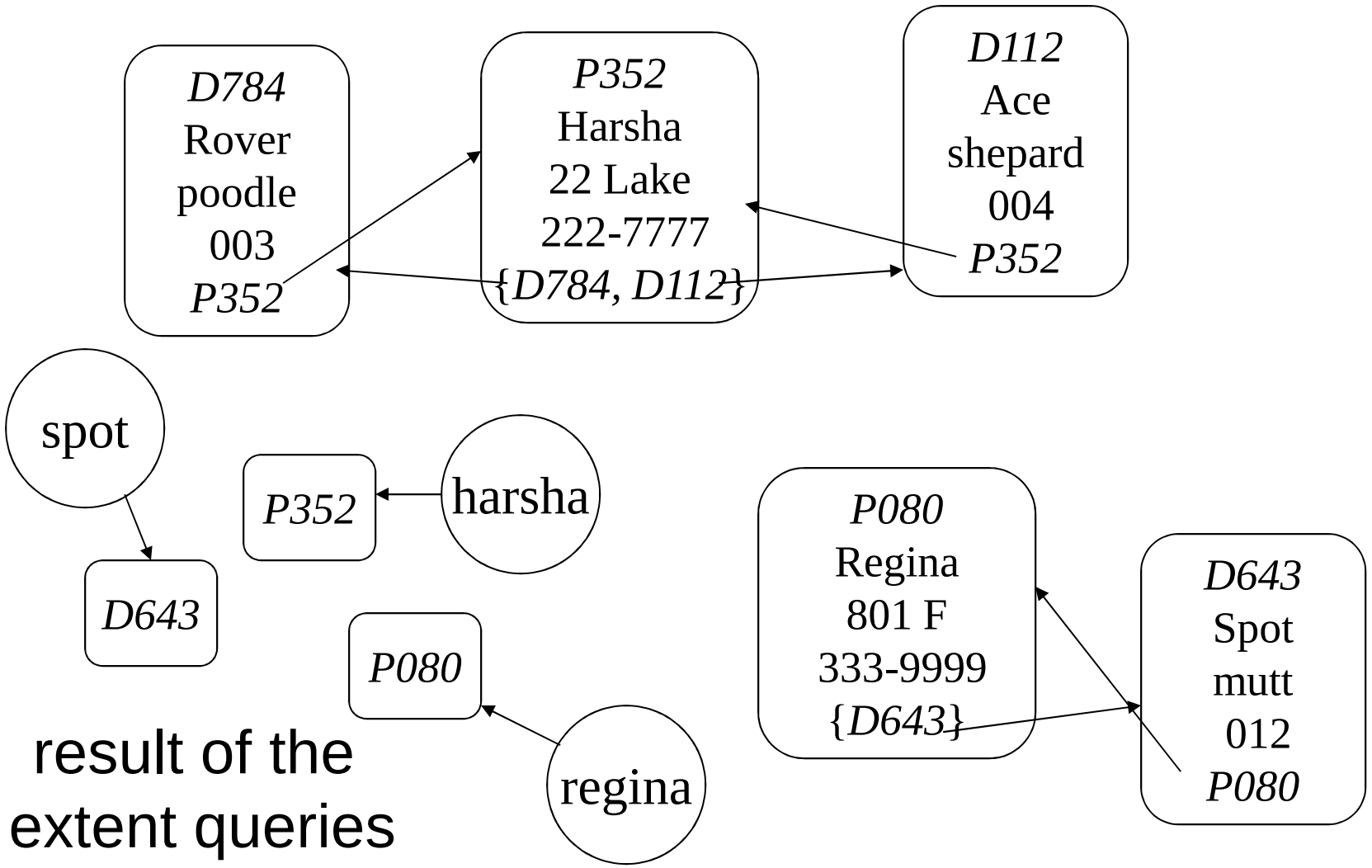


Relationship Maintenance

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

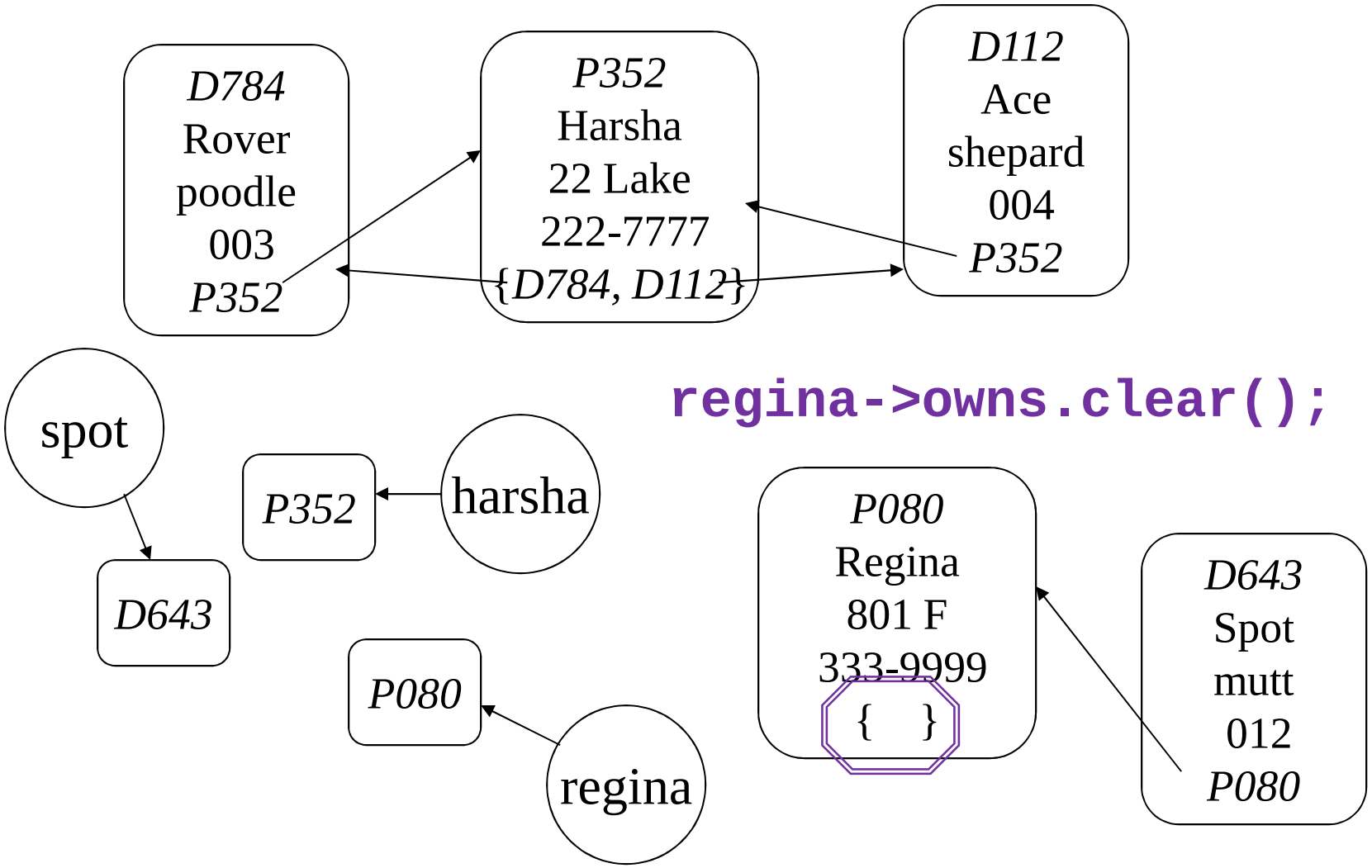
```
d_Ref<Person> regina =
  people.select_element("name='Regina'");
d_Ref<Person> harsha =
  people.select_element("name='Harsha'");
d_Ref<Dog> spot =
  regina->owns.select_element("name='Spot'");
regina->owns.clear();
harsha->owns.insert_element(&spot);
```

Relationship Maintenance

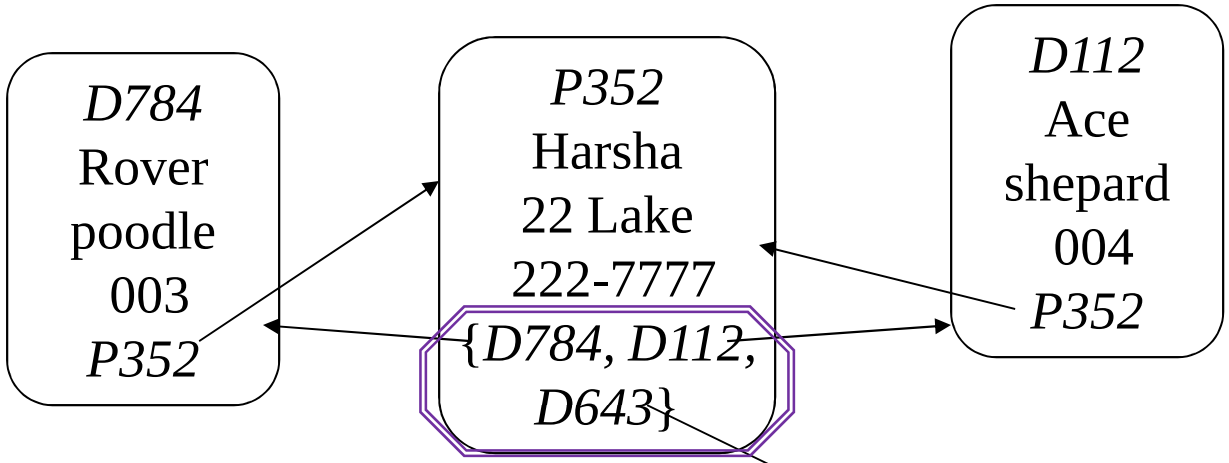


result of the extent queries

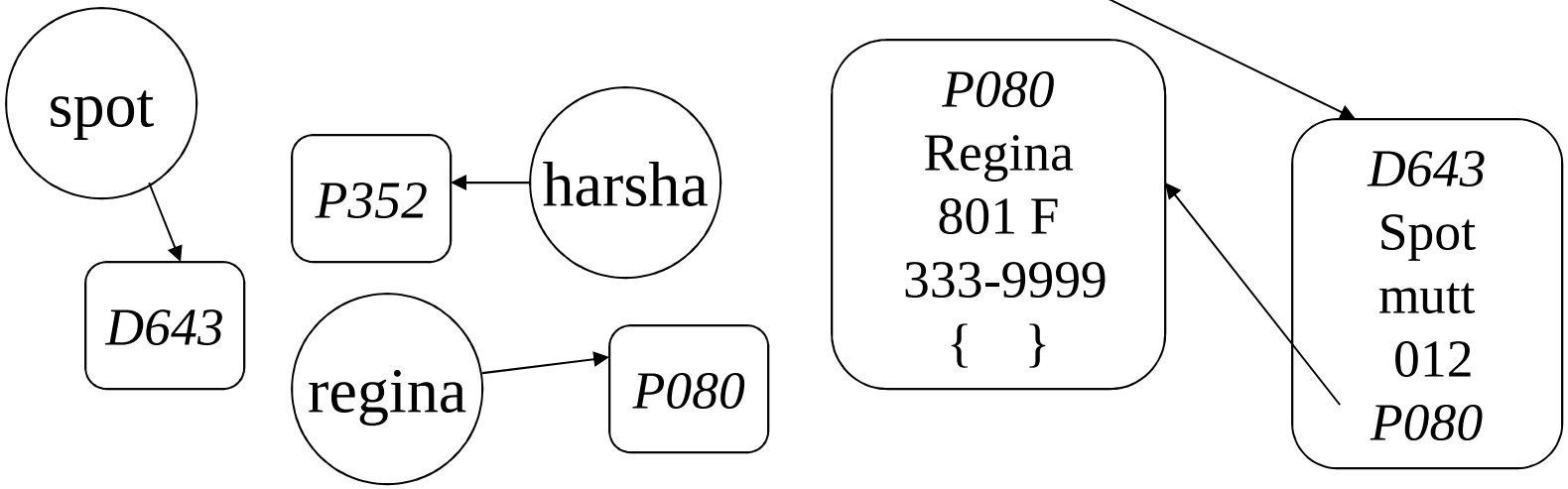
Relationship Maintenance



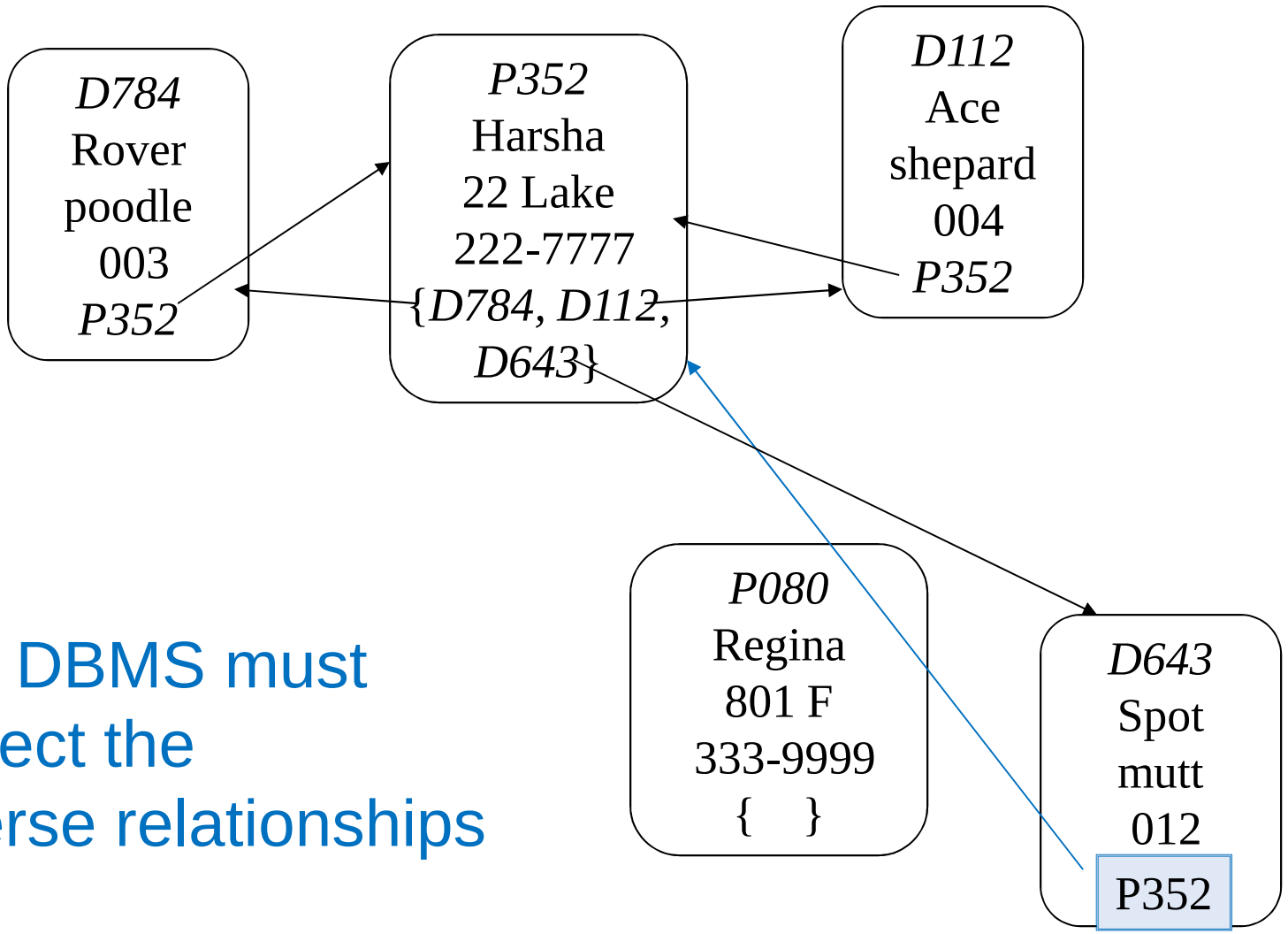
Relationship Maintenance




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



Relationship Maintenance



The DBMS must correct the inverse relationships

OQL

- Schema for OQL examples

```
class Professor
(extent professors, key uid)
{
  attribute string name;
  attribute short uid[9];
  relationship set<Student> advises
    inverse Student::advisor;
  relationship Department dept
    inverse Department::faculty;
  float avgGPAofAdvisees();
};
```

```
class Student
(extent students, key uid)
{
  attribute string name;
  attribute short uid[9];
  relationship Professor advisor
    inverse Professor::advises;
  float GPA();
};
```

OQL example

```
select s.name
from Students s
```

- students is the *extent* of class Student
- s will refer to all Student instance (just as it would refer to all rows in a relation)
- answer is a set of strings

OQL example

```
select s.name
from Students s
where s.gpa() >= 3.0
```

- methods can be applied to objects

OQL example

```
select struct(name: s.name,
              gpa: s.GPA())
from Students s
```

- answer is a set of tuples
- build tuple structure in the query

  (essentially the same as renaming attributes in relational result)
- call to method in select clause

OQL example

```
select s.name
from Professors p,
     p.advises s
where p.name = “Jill Pebble”
  and s.GPA >= 3.0
```

- equivalent to a join
- What does this query compute?

OQL – optimization issues

```
select p.name
from Professors p,
where p.avgGPAofAdvisees() >= 3.0
```

- What objects are accessed by this query?
- Can we estimate the cost of
  an arbitrary user-defined function?

ODMG Database Fucntionality

- d_Database and d_Transaction classes provide basic DBMS functionality
  - open close database connection
  - start/commit/abort transactions
  - retrieve objects by name (essential for persistence/reachability)

d_Database class

```
class d_Database
{
public:
  d_Database();

  enum access_status { not_open, read_write,
                       read_only, exclusive };

  void open(char *database_name,
            access_status _status = read_write);
  void close();

  void set_object_name(d_Ref_Any &theObject, char* theName);
  void rename_object(char *oldName, char *newName);
  d_Ref_Any lookup_object(char * name);

};
```

d_Transaction class

```
class d_Transaction
{
public:
    d_Transaction();
    ~d_Transaction();
    void begin();
    void commit();
    void abort();
    void checkpoint();
};
```

ODMG Extents

- ODL preprocessor generates Set objects that hold all instances of a class:

```
Set<Ref<PERSON> > persons;
Set<Ref<DOG> > pets;
```

- Extents can be accessed with iterators:

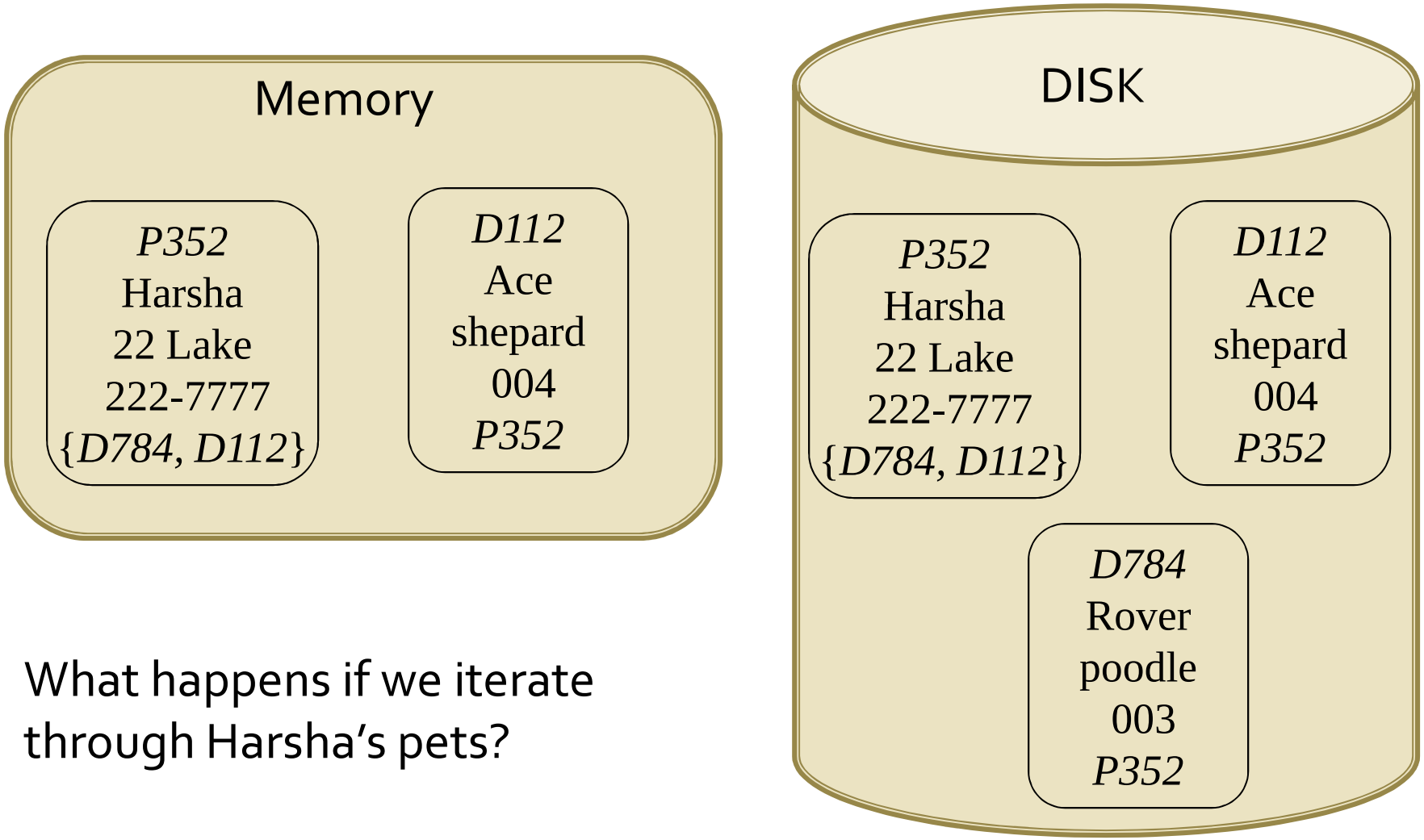
```
d_Iterator<d_Ref<DOG> > pet_iter =
            pets.create_iterator();

d_Ref<DOG> p;
while (pet_iter.next(p))
      cout << p->name;
```

Persistence Capability

- All classes inherit from the system defined class **d_Object**
- Defines everything that DBMS needs make object behave as a database object.
  - object identifier (OID)
  - object size
  - modified or dirty flag: indicates if an object's state has changed from the value currently stored on disk
  - locking information: controls access to objects from multiple applications
  - object's class

Buffer Management



What happens if we iterate through Harsha's pets?

Smart Pointers

```
d_Iterator<d_Ref<DOG> > iter =
          harsha.owns.create_iterator();
d_Ref<DOG> p;
while (pet_iter.next(p)) cout << p->name;
```

- at some point, p will refer to Rover, who is still on disk
- d_Ref will recognize this and read object from disk before returning pointer

What does this mean in terms of our discussions of disk access costs?