Intro to DB

CHAPTER 9
OBJECT-BASED DATABASES

Chapter 9: Object-Based Databases

- Complex Data Types
- Structured Data Types and Inheritance in SQL
- Table Inheritance
- Array and Multiset Types in SQL
- Object Identity and Reference Types in SQL
- Implementing O-R Features
- Persistent Programming Languages
- Object-Oriented vs Object-Relational Databases

Need for Complex Data Types

- Traditional database applications had simple data types
  - Relatively few data types, first normal form
- Complex data types have grown more important in recent years
  - E.g. Addresses can be viewed as a
    - Single string, or
    - Separate attributes for each part, or
    - Composite attributes (which are not in first normal form)
  - E.g. it is often convenient to store multivalued attributes as-is, without creating a separate relation to store the values in first normal form
- Applications
  - computer-aided design, computer-aided software engineering
  - multimedia and image databases, and document/hypertext databases.

Object Structure

- Loosely speaking, an **object** corresponds to an entity in the E-R model.
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into single unit.
- An object has:
  - A set of **variables** that contain the data for the object. The value of each variable is itself an object.
  - A set of **messages** to which the object responds; each message may have zero, one, or more *parameters.*
  - A set of **methods**, each of which is a body of code to implement a message; a method returns a value as the *response* to the message

Class Definition Example

```
class employee {
        /*Variables */
            string    name;
            string    address;
            date      start-date;
            int       salary;
        /* Messages */
            int       annual-salary();
            string    get-name();
            string    get-address();
            int       set-address(string new-address);
            int       employment-length();
};
```

- Methods to read and set the other variables are also needed with strict encapsulation
- Methods are defined separately
  - E.g. **int** *employment-length*() { **return** *today*() – *start-date*; }
    int *set-address*(**string** *new-address*) { *address* = *new-address*;}

Object Classes

- Similar objects are grouped into a **class**
- Each individual object is called an **instance** of its class
- All objects in a class have the same
  - variables, with the same types
  - message interface
  - methods

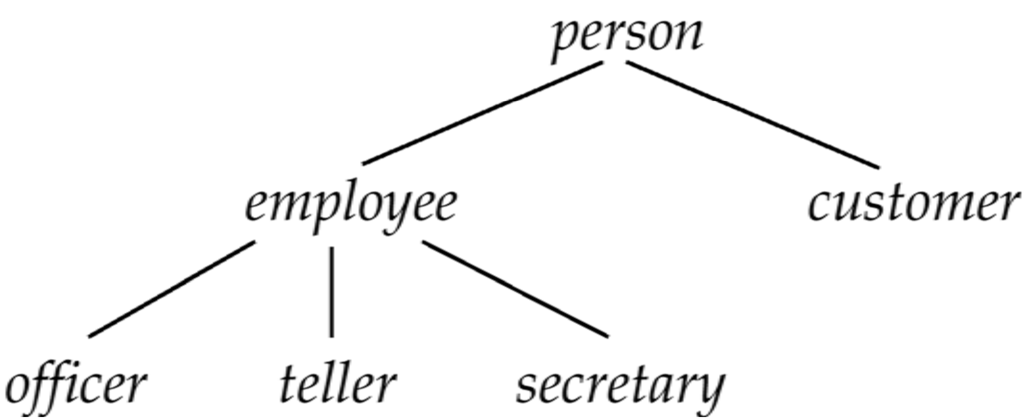
  They may differ in the values assigned to variables
- e.g., group objects for people into a *person* class
- Classes are analogous to entity sets in the E-R model

Inheritance

- Class of bank customers VS class of bank employees
  - Share some variables and messages, e.g., *name* and *address.*
  - But others are specific to each class
    - e.g., *salary* for employees and *credit-rating* for customers.
- Employee and customer are persons
  - every employee is a person; thus *employee* is a specialization of *person*
  - *customer* is a specialization of *person.*
- Create classes *person, employee* and *customer*
  - variables/messages applicable to all persons => associate with class *person.*
  - variables/messages specific to employees => associate with class *employee*
  - similarly for *customer*

Inheritance (Cont.)

- Place classes into a specialization/IS-A hierarchy
  - variables/messages belonging to class *person* are *inherited* by class *employee* as well as *customer*
- Result is a **class hierarchy** (or inheritance hierarchy)



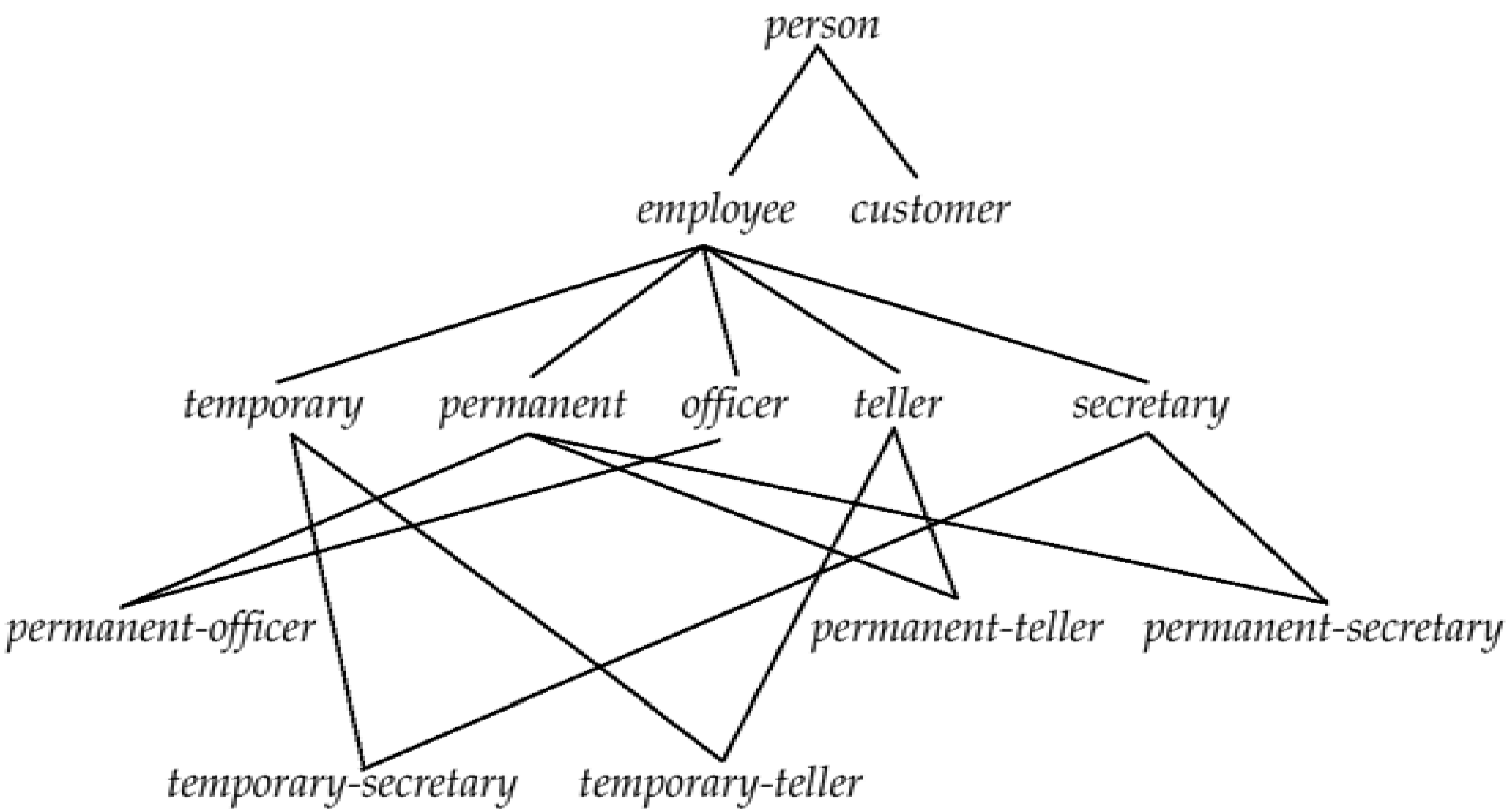
Note analogy with ISA Hierarchy in the E-R model

Class Hierarchy Definition

```
class person{
        string      name;
        string      address:
        };
class customer isa person {
        int credit-rating;
        };
class employee isa person {
        date start-date;
        int salary;
        };
class officer isa employee {
        int office-number,
        int expense-account-number,
        };
        .
        .
        .
```

Multiple Inheritance (Example)

Class DAG for banking example.



Multiple Inheritance

- A class may have more than one superclass.
  - Represented by a **directed acyclic graph** (**DAG**)
  - Particularly useful when objects can be classified in more than one way
- A class inherits variables and methods from *all* its superclasses
- Potential for ambiguity
  - when a variable/message N with the same name is inherited from two superclasses A and B
  - No problem if the variable/message is defined in a shared superclass
  - Otherwise, do one of the following
    - flag as an error,
    - rename variables (A.N and B.N)
    - choose one.

Object Identity

- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.
- Object identity is a stronger notion of identity than in programming languages or other data models
- Identity by
  - Value – data value; e.g. primary key value used in relational systems.
  - Name – supplied by user; file names in UNIX
  - Built-in – identity built into data model or programming language.
    - no user-supplied identifier is required
    - the form of identity used in object-oriented systems

Object Identifiers

- **Object identifiers** are used to uniquely identify objects
- Object identifiers are unique:
  - no two objects have the same identifier
  - each object has only one object identifier
- Can be stored as a field of an object, to refer to another object.
  - E.g., the *spouse* field of a *person* object may be an identifier of another *person* object.
- Can be
  - system generated (created by database) or
  - external (such as social-security number)
- System generated identifiers:
  - Are easier to use, but cannot be used across database systems

Object-Relational Model

- Extended relational model to support
  - Nested relations
  - Complex types
  - Object orientation
- Most commercial DBMS claim to be OR
  - Oracle, DB2, Informix, …
- Relational model
  - First Normal Form: all attributes have atomic domains
- Nested relational model
  - Domains may be atomic or *relation-valued*
    - tuple (complex structure)
    - set (multiset)

Example of a Nested Relation

- Example: library information system
- Each book has
  - title,
  - a set of authors,
  - publisher, and
  - a set of keywords

- Non-1NF relation *books*

title	*author-set*	*publisher*	*keyword-set*
		(name, branch)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

1NF Version of Nested Relation

- 1NF version of *books*

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

flat-books

Decomposition

- Dependencies in *doc*

  title $\rightarrow\rightarrow$ *author* (MVD)

  title $\rightarrow\rightarrow$ *keyword*

  title $\rightarrow$ *pub_name, pub_branch*

- Decomposed version
  - 4NF (BCNF extended to include MVD)
  - Loose 1-to-1 correspondence between a tuple and a doc

title	*author*
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub-name*	*pub-branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

MVD: multi-valued dependency; X $\rightarrow\rightarrow$ Y means that a set of Y values is associated with each X value

Complex Types and SQL:1999

- Extensions to SQL to support complex types include:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- Our description is mainly based on the SQL:1999 standard
  - Not fully implemented in any database system currently
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports

Structured Types and Inheritance in SQL

- Structured types can be declared and used in SQL

  create type *Name* **as**
  (first*name* **varchar**(20),
  lastname **varchar**(20))
  final

  create type *Address* **as**
  (*street* **varchar**(20),
  city **varchar**(20),
  zipcode **varchar**(20))
  not final

  - Note: **final** and **not final** indicate whether subtypes can be created
- Structured types can be used to create tables with composite attributes

  create table *customer* (
  name *Name,*
  address *Address,*
  dateOfBirth **date**)
- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- User-defined row types

  create type *CustomerType* **as** (

  name Name,

  address Address,

  dateOfBirth **date**)

  not final

- Can then create a table whose rows are a user-defined type

  create table *customer* **of** *CustomerType*

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)

  returns interval year

- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)

  returns interval year

  for *CustomerType*

  begin

  return *onDate* - **self**.*dateOfBirth*;

  end

- We can now find the age of each customer:

  select *name.lastname, ageOnDate* (**current_date**)

  from *customer*

Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
  under Person
  (degree       varchar(20),
   department  varchar(20))
create type Teacher
  under Person
  (salary        integer,
   department  varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Inheritance

- SQL:1999 and SQL:2003 do not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*
  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*
  under
  Student **with** (*department* **as** *student_dept*),
  Teacher **with** (*department* **as** *teacher_dept*)

Object-Identity and Reference Types

- An attribute can be a reference to a tuple in a table
- Define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person,* with table *people* as scope:

  create type *Department* (
  name **varchar** (20),
  head **ref** (*Person*) **scope** *people*)

- We can then create a table *departments* as follows

  create table *departments* **of** *Department*

- We can omit the declaration **scope** people from the type declaration and instead make an addition to the **create table** statement:

  create table *departments* **of** *Department*
  (*head* **with options scope** *people*)

Initializing Reference-Typed Values

- To create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately:

insert into *departments*
    values (`CS', null)
update *departments*
    set *head* = (**select** *p.person_id*
            from *people* **as** *p*
            where *name* = `John')
    where *name* = `CS'

Path Expressions

- Dot (.) notation is used for composite attributes

```
select title, publisher.name
from books
```

- Pointer (->) notation is used for reference attributes

```
select head->name, head->address
from departments
```

  - references can be used to hide join operations

Collection-Valued Attributes

- Can be treated much like relations, using the keyword **unnest**
  - The *books* relation has array-valued attribute *author-array* and set-valued attribute *keyword-set*
- Find all books that have the word "database" as keyword

  select *title*
  from *books*
  where 'database' **in** (**unnest**(*keyword-set*))

  - Note: the only collection type supported by SQL:1999 is the array type
- To get a relation containing pairs of the form "title, author-name" for each book and each author of the book

  select *B.title, A*
  from *books* **as** *B*, **unnest**(*B.author-array*) **as** *A*

Collection Valued Attributes (Cont.)

- We can access individual elements of an array by using indices
  - E.g. If we know that a particular book has three authors, we could write:

    select *author-array*[1], *author-array*[2], *author-array*[3]
    from *books*
    where *title* = `Database System Concepts'

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes

select *title*, *A* **as** *author*, *publisher.name* **as** *pub_name*,
publisher.branch **as** *pub_branch*, *K* **as** *keyword*

from *books* **as** *B*, **unnest**(*B.author-array*) **as** *A*,
unnest (*B.keyword-list*) **as** *K*

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Nesting

- Opposite of unnesting: creates a collection-valued attribute
  - NOTE: SQL:1999 does not support nesting
- Similar to aggregation, but using the function **set**()

select *title*, *author*, Publisher(*pub_name, pub_branch*) **as** *publisher*,
  set(*keyword*) **as** *keyword-list*
from *flat-books*
group by *title, author, publisher*

select *title*, **set**(*author*) **as** *author-list*,
  Publisher(*pub_name, pub_branch*) **as** *publisher*,
  set(*keyword*) **as** *keyword-list*
from *flat-books*
group by *title*, *publisher*

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the select clause.

select *title*,
  (**select** *author*
   from *flat-books* **as** *M*
   where *M.title*=*O.title*) **as** *author-set*,
  Publisher(*pub-name, pub-branch*) **as** *publisher*,
  (**select** *keyword*
   from *flat-books* **as** *N*
   where *N.title* = *O.title*) **as** *keyword-set*
from *flat-books* **as** *O*

Nesting & Unnesting

- Unnesting

  select *title, A* **as** *author, publisher.name* **as** *pub_name, publisher.branch* **as** *pub_branch, K* **as** *keyword*
  from *doc* **as** *B,* **unnest**(*B.author_list*) **as** *A,* **unnest**(*B.keyword_set*) **as** *K*

  - result is *flat_books*

- Nesting

  select *title, author,* (*pubname, pubbranch*) as *publisher,* **set**(*keyword*) **as** *keyword_list*
  from *flat_docs*
  group by *title, author, publisher*

  - result is shown below

title	*author*	*publisher*	*keyword-set*
		(*pub-name, pub-branch*)	
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

Object-Oriented Languages

- Object-oriented concepts can be used in different ways
- Object-orientation can be used as a design tool
  - and then encode into, for example, a relational database
  - analogous to modeling data with E-R diagram and then converting to a set of relations
- Object orientation can be incorporated into a programming language that is used to manipulate the database.
  - **Object-relational systems** – add complex types and object-orientation to relational language
  - **Persistent programming languages** – extend object-oriented programming language to deal with databases by adding concepts such as persistence and collections.

Persistent Programming Languages

- Persistent Programming languages
  - allow objects to be created and stored in a database,
  - and used directly from a programming language
- Allow data to be manipulated directly from the programming language
  - No need to go through SQL
  - No need for explicit format (type) changes
- Drawbacks
  - Flexibility and power of programming languages => it is easy to make programming errors that damage the database
  - Complexity of languages makes automatic optimization more difficult
  - Do not support declarative querying as well as relational databases

OO vs OR

- OO
  - efficient in complex main memory operations of persistent data
  - susceptible to data corruption
- OR
  - declarative and limited power of (extended) SQL (compared to PL)
  - data protection and good optimization
  - extends the relational model to make modeling and querying easier
- Relational
  - simple data types, good query language, high protection

END OF CHAPTER 9