Chapter 22: Object-Based Databases

Database System Concepts, 6th Ed.

Chapter 22: Object-Based Databases

- Introduction and Motivation
- Object-relational and Object-oriented Databases
- Complex Data Types and Object Orientation
- 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
- Comparison of Object-Oriented and Object-Relational Databases

Introduction

- Traditional database applications consist of data-processing tasks with relatively simple data types (char, varchar, number, date, blob, clob etc.) which are well suited to the relational data modes. Examples: banking and payroll management.
- Database systems with wider range of applications, such as computer aided design (CAD) and geographical information system (GIS) face limitations imposed by the relational model.
- The solution is to introduce object-based databases, which allow one to deal with complex data types.

Overview / Motivation

- The **first** obstacle faced by programmers using relational data model was the limited type system supported by the relational model.
- Complex application domains requires correspondingly complex data types, such as nested record structures, multivalued attributes and inheritance, which are supported by traditional programming languages such as C++ and Java. Such features are in fact supported in E-R and extended E-R notations, but had to be translated to simpler SQL data types.
- The **object-relational data model** extends the relational data model by providing a richer data types and object orientation. Relational query languages, like SQL, need to be correspondingly extended to deal with the richer type system. Such extensions attempt to preserve the relational foundations while extending the modeling power.
- **Object-relational database systems**, that is, database systems based on object- relational model, provide a convenient migration path for users of relational databases who wish to use object-oriented features.

Overview (Cont.)

- The **second** obstacle was the difficulty in accessing database data from programs written in programming languages such as C++ or Java. Merely extending the type system supported by the database was not enough to solve this problem completely.
- Differences between the type system of the database and the type system of the programming language make data storage and retrieval more complicated and need to be minimized.
- Having to express database access using a language (SQL) which is different from programming language again makes the job of the programmer harder.
- It is desirable, for many applications, to have programming language constructs or extensions that permit direct access to data in the database, without having to go through an intermediate language like SQL.

Persistent Programming Language

- **Persistent programming language** refers to extensions of existing programming languages to add persistence and other database features, using the native type system of the programming language.
- **Object-oriented database system** refers to database systems that support an object-oriented type system and allow direct access to data from an object-oriented programming language using the native type system of the language.

Object-Based Data Models

Object-Relational Data Model

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Object-Oriented Data Model

- Supports an object-oriented type system
- Allows direct access to data from an object-oriented programming language using the native type system of the language.

Complex Data Types

- Traditional database applications have conceptually simple data types.
  - The basic data items are records that are fairly small and whose fields are atomic – that is, they are not further structured.
  - First normal form holds
  - There are only few record types
- In recent years, demand has grown for ways to deal with more complex data types.
  - Composite Vs component attributes – address (street_ address, city, state, postal code), name (first name, middle name, last name).
  - While an entire address could be viewed as an atomic data item of type string, this view would hide details which could be of interest to queries. On the other hand, if an address were represented by breaking it into the components, writing queries would be more complicated since they would have to mention each field.

Complex Data Types (Cont.)

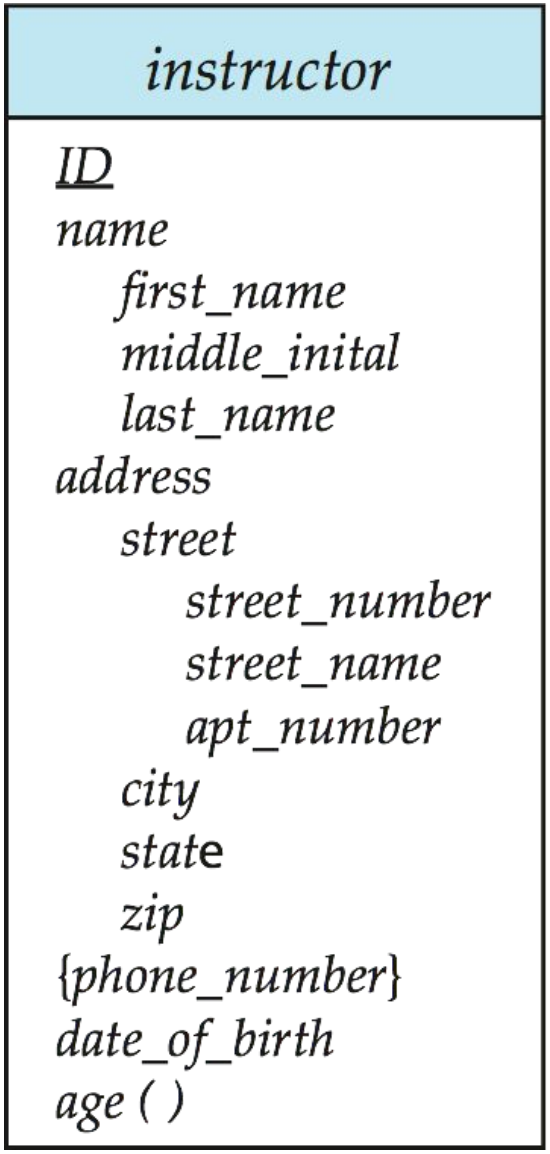
- A better alternative to allow structured data types, which allow a type *address* with subparts *street_address, city, state* and *postal code*.
- Single Vs Multple Valued attribute: Consider multivalued attributes from E-R model such as phone number. The alternative of normalization by creating a new relation is expensive and artificial.

- With complex type systems we can represent E-R model concepts, such as composite attributes, multivalued attributes, generalization and specialization directly, without a complex translation to the relational model.
- First normal form (1NF) requires that all attributes have *atomic domains* and a domain is *atomic* if elements of the domain are considered to be individual units.
- The assumption of 1NF is natural one in the bank examples. However, not all applications are best modeled by 1NF relations. For example, rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities). These objects may require several records for their representation.

Complex Data Types

- Motivation:
  - Permit non-atomic domains (atomic ≡ indivisible)
  - Example of non-atomic domain: set of integers,or set of tuples
  - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
  - allow relations whenever we allow atomic (scalar) values — relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form.

Simple, Composite, Multivalued and Derived Attributes



Example of a Nested Relation

- Example: library information system
- Each book has
  - title,
  - a list (array) of authors,
  - Publisher, with subfields *name* and *branch*, and
  - a set of keywords
- Non-1NF relation *books*

title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

Nested Relation and 1NF version

title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

Non 1NF version of relation *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

1NF version of relation *books: flat_books*

Example of a Non-1NF Relation

- **Authors.** A book may have a list of authors, which can be represented as an array. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "Authors".
- **Keyword.** If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or mode specific keywords. Thus, we view the domain of the set of keywords as non-atomic.
- **Publisher.** Unlike *keywords* and *authors*, *publisher* does not have a set-valued domain. However, we may view *publisher* as consisting of the subfields *name* and *branch*. This view makes the domain of *publisher* non-atomic.

4NF Decomposition of Nested Relation

- Suppose for simplicity that title uniquely identifies a book
  - In real world ISBN is a unique identifier
- Decompose *books* into 4NF using the schemas:
  - (*title, author, position*)
  - (*title, keyword*)
  - (*title, pub-name, pub-branch*)
- 4NF design requires users to include joins in their queries.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

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

Problems with 4NF Schema

- 4NF design requires users to include joins in their queries.
- 1NF relational view *books* defined by join of 4NF relations:
  - eliminates the need for users to perform joins,
  - but loses the one-to-one correspondence between tuples and documents.
  - And has a large amount of redundancy
- Nested relations representation is much more natural here.

- The ability to use complex data types such as sets and arrays can be useful in many applications but should be used with care.

Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
  - Collection and large object types
    - Nested relations are an example of collection types
    - Arrays and multisets are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- 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** (a.k.a. **user-defined types**) can be declared and used in SQL

```
create type Name as
 (firstname        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 person (
   name       Name,
   address    Address,
   dateOfBirth date)
```

- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- **User-defined row types**

  create type *PersonType* **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*

Structured Types (cont.)

- Alternative using **unnamed row types**.

```
create table person_r(
    name        row(firstname varchar(20),
                    lastname varchar(20)),
    address     row(street    varchar(20),
                    city      varchar(20),
                    zipcode   varchar(20)),
    dateOfBirth date)
```

- This definition is equivalent to the preceding table definition, except that the attributes *name* and *address* have unnamed types and rows of the table also have an unnamed type.
- To find the last name and city of each customer, we write a query

```
select name.lastname, address.city
from customer
```

Methods

- A structured type can have methods defined on it. We declare methods as part of type definition of a structured type.

create type *CustomerType* **as** (

name Name,

address Address,

dateOfBirth **date**)

not final

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

returns interval year

Methods (Cont.)

- Method body is given separately.

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

  returns interval year

  for *CustomerType*

  begin

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

  end

  - **for** clause indicates which type this method is for
  - The keyword **instance** indicates that this method executes on an instance of the *Customer* type
  - The variable **self** refers to the *Customer* instance on which the method is invoked.
- Methods can be invoked on instances of a type. We can now find the age of each customer:

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

  from *customer*

Constructor Function

- In SQL:1999 **constructor functions** are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type. For instance, we could declared a constructor for the type *Name* like this:

```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self. firstname = firstname;
    set self. lastname = lastname;
end
```

- We can then use **new** *Name* ('John', 'Smoth') to create a value for the type *Name*.
- By default, every structured type has a constructor with no argument, which sets the attributes to their default values. Any other constructors have to be created explicitly.

Constructor Function (Cont.)

- There can be more than one constructor for the same structured type; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments.
- The following statement illustrates how we can create a new tuple in the *Customer* relation. We assume that a constructor has been defined for *Address*, just like *Name*.

  Insert into Customer

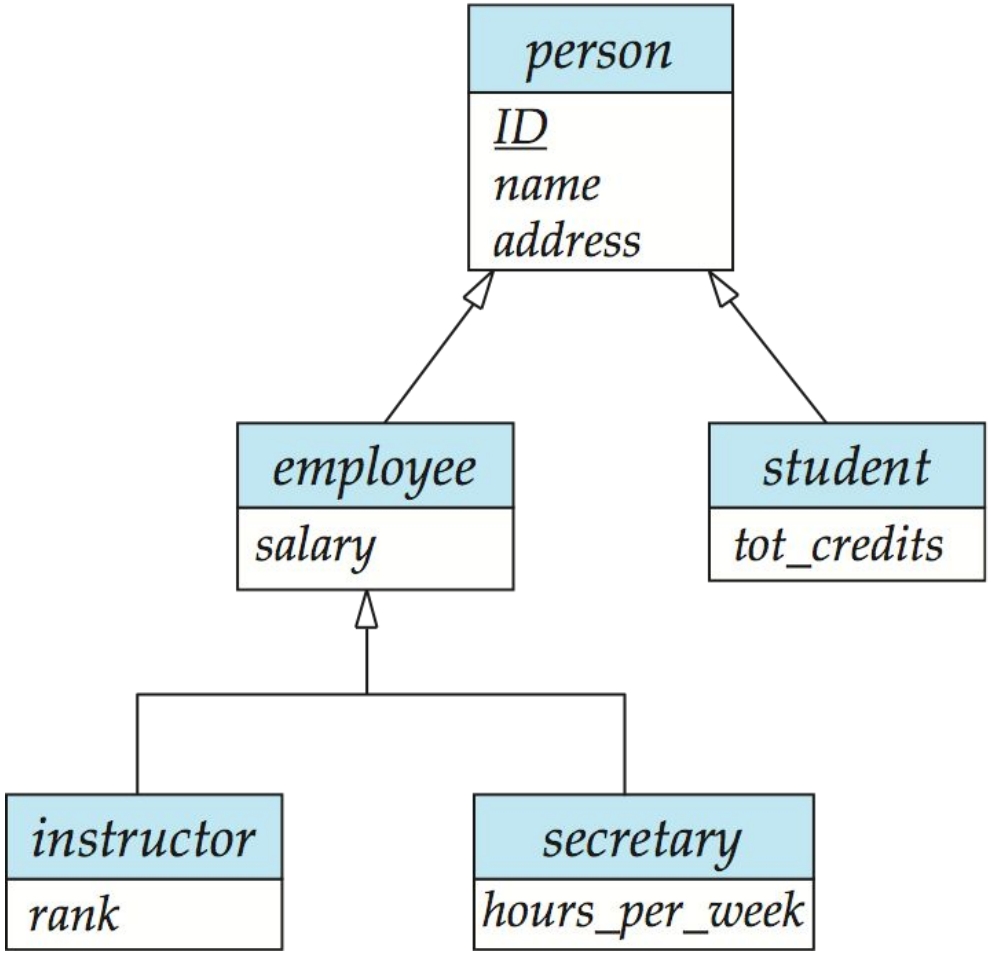
  values

  (**new** *Name* (‘John’, ‘Smith’),

  new *Address* (’20 Main St’, ‘New York’, ‘11001’),

  date ‘1960-8-22’)

Type Inheritance: Specialization in E_R Model



Type Inheritance

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

```
create type Person
    (ID varchar(10), name varchar(20),
        address varchar(20))
```

- We may want to store extra information about people who are students and about people who are teachers.
- Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:

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

- Both *Student* and *Teacher* inherit the attributes of *Person* – namely, *ID*, *name* and *address*. *Student* and *Teacher* are said to be subtypes of *Person* and *Person* is a supertype of *Student* as well as *Teacher*.
- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Type Inheritance

- If we want to store information about *teaching_assistant*, who are simul-taneously students and teachers, perhaps even in different departments; this can be done by 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*

- *Teaching_Assistant* would inherit all the attributes of *Student* and *Teacher*. But there is a problem since the attributes *ID*, *name*, *address* and *department* are present in both *Student* and *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*)

- SQL:1999 and SQL:2003 do not support multiple inheritance. **SQL supports only single inheritance** – a type can inherit from only a single type.

Table Inheritance

- Tables created from subtypes can further be specified as **subtables.** subtables in SQL correspond to the E-R notion of specialization and generalization.
- E.g. **create table** *people* **of** *Person;*
  create table *students* **of** *Student* **under** *people;*
  create table *teachers* **of** *Teacher* **under** *people;*
- Tuples added to a subtable are automatically visible to queries on the supertable
  - E.g. query on *people* also sees *students* and *teachers*.
  - Similarly updates/deletes on *people* also result in updates/deletes on subtables (**delete from** *people* **where** P)
  - To override this behaviour, use "**only** *people*" in query
- Conceptually, multiple inheritance is possible with tables
  - e.g. *teaching_assistants* under *students* and *teachers*
  - *But is not supported in SQL currently*
    4. So we cannot create a person (tuple in *people*) who is both a student and a teacher

Consistency Requirements for Subtables

- There are some consistency requirements for subtables. A definition says ‘tuples in a subtable **correspond** to tuples in a parent table if they have the same values for all inherited attributes’.
- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of its immediate subtables (e.g. *students* and *teachers)*
  - Additional constraint in SQL:1999: *Disjoint NOT Overlapping!*

    All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
    - That is, each entity must have a most specific type
    - We cannot have a tuple in *people* corresponding to a tuple each in *students* and *teachers*
    - *NO multiple inheritance!*

Array and Multiset Types in SQL

- SQL supports two *collection* types:
  - arrays (SQL:1999) and multisets (SQL:2003).
  - A *multiset* is an unordered collection, where an element may occur multiple times. Multisets are like sets, except that a set allows each element to occur at most once.
- Suppose we wish to record information about books, including a set of keywords for each book. Also we wished to store the names of authors of a book as an array;
- Unlike elements in a multiset, the elements of an array are ordered, so we can distinguish the first author from the second author and so on.

Array and Multiset Types in SQL

- Example of array and multiset declaration:

```
create type Publisher as
  (name          varchar(20),
   branch        varchar(20));

create type Book as
   (title          varchar(20),
    author_array   varchar(20) array [10],
    pub_date       date,
    publisher      Publisher,
    keyword-set    varchar(20) multiset);

create table books of Book;
```

- In general, multivalued attributes from an E-R schema can be mapped to multiset-valued attributes in SQL; if ordering is important, SQL arrays can be used instead of multisets.

Creation and Accessing Collection Values

- An array of values can be created in SQL:1999 in this way:

  array [‘Silberschatz’,`Korth’,`Sudarshan’]

- A multisets of keywords can be constructed as follows:

  multiset [‘computer’, ‘database’, ‘SQL’]

- To create a tuple of the type defined by the *books* relation:
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’])

  Here we have created a value for *Publisher* by invoking a constructor function for *Publisher* with appropriate arguments.

- To insert the preceding tuple into the relation *books*

  insert into *books*
  values
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’])

  We can access or update elements of an array by specifying the array index, for example author_array[1]

Querying Collection-Valued Attributes

- To find all books that have the word "database" as one of the keywords,

  select *title*
  from *books*
  where 'database' **in** (**unnest**(*keyword_set*))

- 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'

- 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.author*
  from *books* **as** *B*, **unnest** (*B.author_array*) **as** *A* (*author*)

- When unnesting an array, the previous query loses information about the ordering of elements in the array. To retain ordering information we add a **unnest with ordinality** clause

  select *B.title, A.author, A.position*
  from *books* **as** *B*, **unnest** (*B.author_array*) **with ordinality as**
  A (*author, position*)

title	*author_array*	*publisher* (*name, branch*)	*keyword_set*
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

title	author
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

Title	Author	Position
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

title	author	pub_name	pub_branch	keyword
Compilers	[Smith, Jones]	McGraw-Hill	New York	parsing
Compilers	[Smith, Jones]	McGraw-Hill	New York	analysis
Networks	[Jones, Frick]	Oxford	London	Internet
Networks	[Jones, Frick]	Oxford	London	Web

title	author	pub_name	pub_branch	keyword
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}

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**.
- E.g.

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

- Result relation *flat_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

Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- Nesting can be done in a manner similar to aggregation, but using the function **collect**() in place of an aggregation operation, to create a multiset
- To nest the *flat_books* relation on the attribute *keyword*:

select *title*, *author*, *Publisher* (*pub_name, pub_branch*) **as** *publisher*,
collect (*keyword*) **as** *keyword_set*
from *flat_books*
group by *title, author, publisher*

title	*author*	*publisher* (*pub_name,pub_branch*)	*keyword_set*
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}

- To nest both attribute *author* and *keyword*:

select *title*, **collect** (*author*) **as** *author_set*,
Publisher (*pub_name, pub_branch*) **as** *publisher*,
collect (*keyword*) **as** *keyword_set*
from *flat_books*
group by *title*, *publisher*

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the **select** clause, starting from the 4NF relation *books4*

```
select title,
   array (select author
           from authors as A
           where A.title = B.title                                    order by
A.position) as author_array,
   Publisher (pub_name, pub_branch) as publisher,
   multiset (select keyword
             from keywords as K
             where K.title = B.title) as keyword_set
from books4 as B
```

title	author	position
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

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

Object-Identity and Reference Types

- Object-oriented language provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specific type.
- 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*)
- Here the reference is restricted to tuple of the table *people*. The restriction of the **scope** of a reference to tuples of a table is mandatory in SQL and it makes references behave like foreign key.

System-generated Identifiers

- The referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called **self-referential attribute**, by adding a **ref is** clause to the **create table** statement:

  create table *people* **of** *Person*
  ref is *person_id* **system generated**

- Here, *person_id* is an attribute name, not a keyword and **create table** statement specifies that the identifier is generated automatically by the database.

Initializing Reference-Typed Values

- In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query.
- 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'
```


User Generated Identifiers

- An alternative to system-generated identifiers is to allow users to generate identifiers. The type of the object-identifier must be specified as part of the type definition of the referenced table, and
- The table definition must specify that the reference is user generated

```
create type Person
    (name varchar(20)
     address varchar(20))
    ref using varchar(20)

create table people of Person
    ref is person_id user generated
```

- When creating a tuple, we must provide a unique value for the identifier:

```
insert into people (person_id, name, address ) values
    ('01284567', 'John', `23 Coyote Run')
```

- We can then use the identifier value when inserting a tuple into *departments*
  - Avoids need for a separate query to retrieve the identifier:

```
insert into departments
values(`CS', `02184567')
```

User Generated Identifiers (Cont.)

- It is even possible to use an existing primary key value as the identifier:

  create type *Person*
  (*name* **varchar** (20) **primary key**,
  address **varchar**(20))
  ref from (*name*)

  create table *people* **of** *Person*
  ref is *person_id* **derived**

- When inserting a tuple for *departments*, we can then use

  insert into *departments*
  values(`CS',`John')

Path Expressions

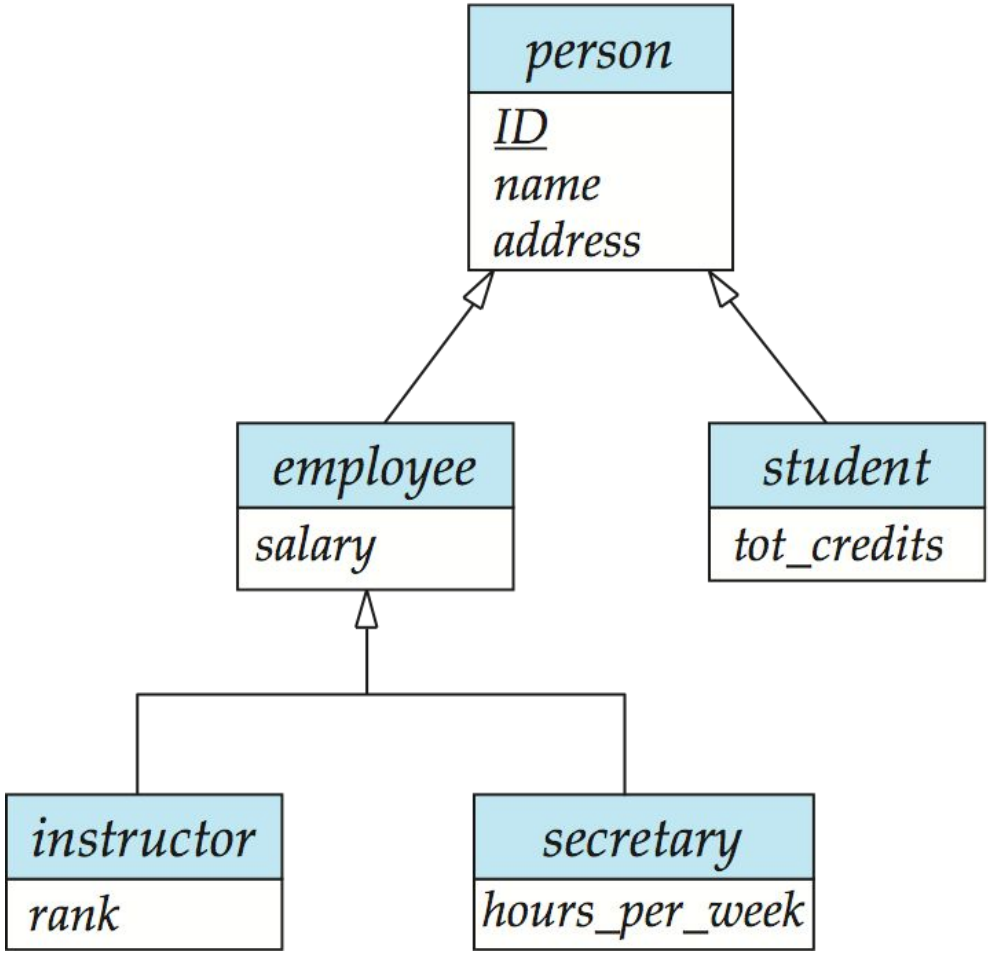
- Find the names and addresses of the heads of all departments:

  select *head* −>*name*, *head* −>*address*
  from *departments*

- An expression such as "head−>name" is called a **path expression**
- Since *head* is a reference to a tuple in the *people* table, the attribute *name* in the preceding query is the *name* attribute of the tuple form the *people* table.
- Path expressions help avoid explicit joins
  - If department head were not a reference, *head* field of *department* would be declared a foreign key of the table *people* and a join of *departments* with *people* would be required to get the address of the head of a department.
  - Makes expressing the query much easier for the user
- We can use the operation **deref** to return the tuple pointed to by a reference and then access its attribute:

  select deref(*head).name*
  from *departments*

Specialization/Generalization Example



Design Constraints on a Specialization/Generalization

- **A. Type1:** Constraint on which entities can be members of a given lower-level entity set.
- **1. Condition-defined:** In condition-defined lower-level entity sets, membership is defined on the basis of whether or not an entity satisfies an explicit condition or predicate. It is also called attribute-defined generalization.
  - Example 1: Saving account and Checking account
  - Example 2: All customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
- **2. User-defined:** Used-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns to a given entity set.
  - Example: Employee and Team

Design Constraints on a Specialization/Generalization

- **B. Type2:** Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
- **1. Disjoint:** A *disjointness constraint* requires that an entity belong to only one lower-level entity set.
  - Noted in E-R diagram by having multiple lower-level entity sets link to the same triangle
  - Example: Saving account and Checking account
- **2. Overlapping:** In *overlapping generalizations*, the same entity may belong to more than one lower-level entity sets within a single generalization.
  - Example: Employee and Team
  - Teaching-assistant

Design Constraints on a Specialization/Generalization

- **C. Completeness constraint:** specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/ specialization.
- **1. Total generalization or specialization:** Each higher-level entity must belong to one of the lower-level entity set.
  - Example: Saving account and Checking account
- **2. Partial generalization or specialization:** Some higher-level entities may not belong to any lower-level entity set.
  - Example: Employee and Team

Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	*ID, name, address*
student	*ID, tot_credits*
employee	*ID, salary*

  - Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	*ID, name, address*
student	*ID, name, address, tot_credits*
employee	*ID, name, address, salary*

  - If specialization is total, the schema for the generalized entity set (*person*) not required to store information
    - Can be defined as a "view" relation containing union of specialization relations
    - But explicit schema may still be needed for foreign key constraints
  - Drawback: *name* and *address* may be stored redundantly for people who are students and/or employees

Implementing O-R Features

- Object-relational database systems are basically extensions of existing relational database systems.
- Changes are clearly required at many levels of the database system. However, to minimize changes to the storage system code (relation storage, indices etc.), the complex data types supported by object-relational systems can be translated to the simpler type system of relational database.
- Similar to how E-R features are mapped onto relation schemas

E-R Model	**Object-Relational Model**
multivalued attribute	multiset-valued attributes
Composite attributes	roughly structured types
ISA hierarchy	table inheritance

- The techniques for converting E-R model features to tables can be used with some extensions to translate object-relational data at the storage level.

Implementing O-R Features

- Subtables can be stored in efficient manner, without replication of all inherited fields, in one of the two ways:
  - 1.1 Each table stores primary key (which may be inherited from a parent table) and the attributes are defined locally.
  - 1.2 Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the supertables, based on the primary key.
  - 2 Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table on which it is inserted and its presence is inferred in each of the supertables. Access to all attributes of a tuple is faster, since a join is not required.
- Implementations may choose to represent array and multiset types directly or may choose to use a normalized representation internally.
- Normalized representation tend to make up more space and require an extra join/grouping cost to collect data in an array or multiset. However, normalized representations may be easier to implement.

Persistent Programming Languages

- Database languages differ from traditional programming languages in that they directly manipulate data that are persistent – data that continue to exist even after program that created it has terminated. Examples of persistent data: a relation in the database and tuples in a relation. In contrast the only persistent data that traditional programming language directly manipulate are files.
- Access to a database is only one component of any real-world appli-cation. While a data manipulation language like SQL is quite effective for accessing data, a programming language is required for implementing other components of the application such as user interfaces or communication with other computers. The traditional way of interfacing database language to programming languages is embedding SQL within the programming language.
- Programming languages that natively and seamlessly allow objects to continue existing after the program has been closed down are called **persistent programming languages**.
- A **persistent programming language** is extended with constructs to handle persistent data (The only commercial product that appears to do this at the moment is **JADE.** Its main competitors are **Java** and **C#)**

Diff. between Persistent Language and embedded SQL

- Persistent programming languages can be distinguished from languages with embedded SQL in at least two ways:
- 1. With an embedded language, the type system with the host language usually differs from the type system of the data manipulation language (SQL-DML).
- The programmer is responsible for any type conversions between the host language and SQL. Having the programmer carry out this task has several drawbacks.
- **Drawbacks:**

  i) Code conversion operates outside of OO type system, and hence has a higher chance of having undetected errors.

  ii) Format conversion takes a substantial amount of code.
- In contrast, in a persistent programming language, the query language is fully integrated with the host language and both share the same type system. Objects can be created and stored in the database without any explicit type or format changes; any format changes required are carried out transparently.

Diff. between Persistent Language and embedded SQL

- 2. The programmer using an embedded SQL is responsible for writing explicit code to fetch data from the database into memory.
- If any updates are performed, the programmer must write code explicitly to store the updated data back in the database.
- In contrast, in a persistent programming language, the programmer can manipulate persistent data without writing codes explicitly to fetch it into memory or store it back to disk.
- **Drawbacks:**

  i) Powerful but easy to make programming errors that damage the database.

  ii) Harder to do automatic high-level optimization and

  iii) Do not support declarative querying well.

Advantages and Drawbacks of Persistent Programming Languages

- Object-oriented programming languages like C++ and Java can be extended to make them persistent programming languages. These language features allow programmers to manipulate data directly from the programming language, without having to go through data manipulation language such as SQL. Thereby, they provide tighter integration of the programming languages with the database than, for example, embedded SQL.
- There are certain drawbacks to persistent programming languages, however, that we must keep in mind when deciding to use them.

  1) Since the programming language is usually a powerful one, it is relatively easy to make programming errors that damage the database.

  2) The complexity of the language makes automatic high-level optimization, such as to reduce disk I/O, harder.

  3) Support for declarative querying is important for many applications, but persistent programming languages currently do not support declarative querying well.

Comparison of O-O and O-R Databases

- **Object-relational database:** This is object-oriented database built on top of the relational model. **Object-oriented database:** This is built around persistent programming languages.
- Persistent extension to programming languages and object-relational systems target different markets:
  - The declarative nature and limited power of the SQL language provides good protection of data from programming errors, and makes high level optimizations, such as reducing I/O, relatively easy.
  - Object-relational systems aim at making data modeling and querying easier by using complex data types.
  - Typical applications include storage and querying of complex data, including multimedia data.

Comparison of O-O and O-R Databases

- A declarative language such as SQL, however, imposes significant performance penalty for certain kinds of applications that run primarily in main memory and that perform a large no. of accesses to the database.
- Persistent programming languages target such applications that have high performance requirements. They provide low overhead access to persistent data, and eliminate the need for data translation if the data are to be manipulated by a programming language.
- However, they are more susceptible to data corruption by programming errors and usually do not have a powerful querying capability.
- Typical application include CAD databases.

Persistent Programming Languages

- Languages extended with constructs to handle persistent data
- Programmer can manipulate persistent data directly
  - no need to fetch it into memory and store it back to disk (unlike embedded SQL)
- Persistent objects:
  - **Persistence by class** - explicit declaration of persistence
  - **Persistence by creation** - special syntax to create persistent objects
  - **Persistence by marking** - make objects persistent after creation
  - **Persistence by reachability** - object is persistent if it is declared explicitly to be so or is reachable from a persistent object

Object Identity and Pointers

- Degrees of permanence of object identity
  - **Intraprocedure**: only during execution of a single procedure
  - **Intraprogram**: only during execution of a single program or query
  - **Interprogram**: across program executions, but not if data-storage format on disk changes
  - **Persistent**: interprogram, plus persistent across data reorganizations
- Persistent versions of C++ and Java have been implemented
  - C++
    - ODMG C++
    - ObjectStore
  - Java
    - Java Database Objects (JDO)

Persistent C++ Systems

- Extensions of C++ language to support persistent storage of objects
- Several proposals, ODMG standard proposed, but not much action of late
  - **persistent pointers**: e.g. d_Ref<T>
  - **creation of persistent objects**: e.g. new (db) T()
  - **Class extents**: access to all persistent objects of a particular class
  - **Relationships:** Represented by pointers stored in related objects
    - Issue: consistency of pointers
    - Solution: extension to type system to automatically maintain back-references
  - **Iterator interface**
  - **Transactions**
  - **Updates:** mark_modified() function to tell system that a persistent object that was fetched into memory has been updated
  - **Query language**

Persistent Java Systems

- Standard for adding persistence to Java : **Java Database Objects (JDO)**
  - Persistence by reachability
  - Byte code enhancement
    - Classes separately declared as persistent
    - Byte code modifier program modifies class byte code to support persistence
      - E.g. Fetch object on demand
      - Mark modified objects to be written back to database
  - Database mapping
    - Allows objects to be stored in a relational database
  - Class extents
  - Single reference type
    - no difference between in-memory pointer and persistent pointer
    - Implementation technique based on **hollow objects** (a.k.a. **pointer swizzling**)

Object-Relational Mapping

- **Object-Relational Mapping (ORM)** systems built on top of traditional relational databases
- Implementor provides a mapping from objects to relations
  - Objects are purely transient, no permanent object identity
- Objects can be retried from database
  - System uses mapping to fetch relevant data from relations and construct objects
  - Updated objects are stored back in database by generating corresponding update/insert/delete statements
- The **Hibernate** ORM system is widely used
  - described in Section 9.4.2
  - Provides API to start/end transactions, fetch objects, etc
  - Provides query language operating direcly on object model
    - queries translated to SQL
- Limitations: overheads, especially for bulk updates

Comparison of O-O and O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, high performance.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system
- Note: Many real systems blur/confuse these boundaries
  - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

End of Chapter 22

Figure 22.05

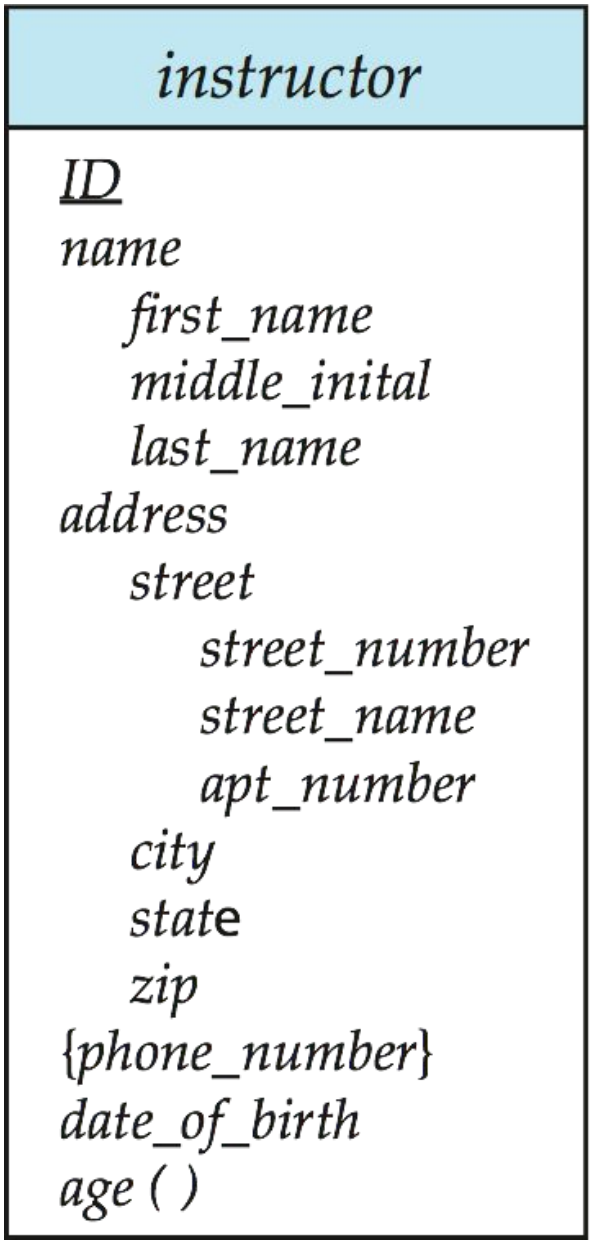


Figure 22.07