Mapeo Objeto Relacional

Merging Relational and Object Models

- Object-oriented models support interesting data types --- not just flat files.
  - Maps, multimedia, etc.
- The relational model supports very-high-level queries.
- Object-relational databases are an attempt to get the best of both.

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:
  - Retains mathematical foundation of relational model
  - Violates first normal form.

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}

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
  - 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

User Defined Types

- A *user-defined type*, or UDT, is essentially a class definition, with a structure and methods.
- Two uses:
  1. As a *rowtype*, that is, the type of a relation.
  2. As the type of an attribute of a relation.

6

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*
- 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)
```

Constructor Functions

- **Constructor functions** are used to create values of structured types
- E.g.

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

- To create a value of type *Name*, we use
  new *Name*('John', 'Smith')
- Normally used in insert statements

```
insert into Person values
    (new Name('John', 'Smith),
     new Address('20 Main St', 'New York', '11001'),
     date '1960-8-22');
```

Type 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 Type 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 )
```

- Each value must have a **most-specific type**

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;
```

Creation of Collection Values

- Array construction
  array [‘Silberschatz’, `Korth’, `Sudarshan’]
- Multisets
  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’])
- 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’]);

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes us called **unnesting**.
- E.g.
  select *title*, *A* **as** *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

Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,
  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*)
- To retain ordering information we add a **with ordinality** clause
  select *B.title*, *A.author*, *A.position*
  from *books* **as** *B*, **unnest** (*B.author_array*) **with ordinality as**
  A (*author*, *position*)

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 **colect**() 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*
groupby *title*, *author*, *publisher*

- To nest on both authors and keywords:

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*

Storing Nested Relations

- Oracle doesn't really store each nested table as a separate relation -- - it just makes it look that way.
- Rather, there is one relation *R* in which all the tuples of all the nested tables for one attribute *A* are stored.
- Declare in CREATE TABLE by:

NESTED TABLE *A* STORE AS *R*

18

Example: Storing Nested Tables

```
CREATE TABLE Manfs (
    name    CHAR(30),
    addr        CHAR(50),
    beers   beerTableType
)
NESTED TABLE beers STORE AS BeerTable;
```

19

References

- If *T* is a type, then REF *T* is the type of a reference to *T*, that is, a pointer to an object of type *T*.
- Often called an "object ID" in OO systems.
- Unlike object ID's, a REF is visible, although it is gibberish.

20

Object-Identity and Reference Types

- 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*)

- Referenced table must have an attribute that stores the identifier, called the **self-referential attribute**

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

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'
```

Object Identifiers Using Reference Types

- **Reference type**
  - Create unique system-generated object identifiers
  - Examples:
    - REF IS SYSTEM GENERATED
    - REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;

User Generated 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.)

- Can 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*

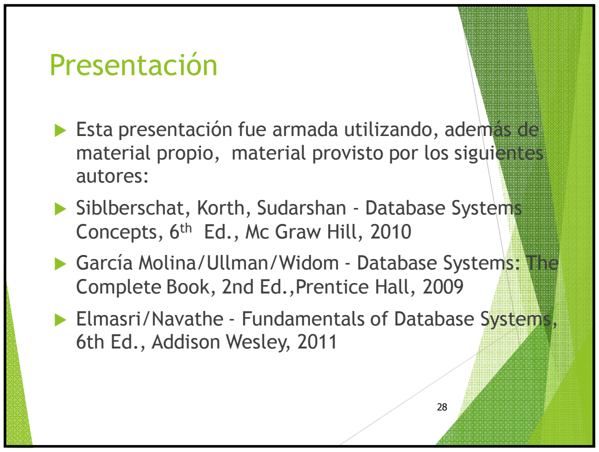
- An expression such as "head->name" is called a **path expression**
- Path expressions help avoid explicit joins
  - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
  - Makes expressing the query much easier for the user

Implementing O-R Features

- Similar to how E-R features are mapped onto relation schemas
- Subtable implementation
  - Each table stores primary key and those attributes defined in that table

  or,

  - Each table stores both locally defined and inherited attributes


Presentación
Esta presentación fue armada utilizando, además de material propio, material provisto por los siguientes autores:
Siblberschat, Korth, Sudarshan - Database Systems Concepts, 6th Ed., Mc Graw Hill, 2010
García Molina/Ullman/Widom - Database Systems: The Complete Book, 2nd Ed.,Prentice Hall, 2009
Elmasri/Navathe - Fundamentals of Database Systems, 6th Ed., Addison Wesley, 2011
28