



# Chapter 22: Object-Based Databases

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Chapter 22: Object-Based 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



# Object-Relational Data Models

- 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.



# Complex Data Types

- Motivation:
  - Permit non-atomic domains (atomic  $\equiv$  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.



# 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*

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



# 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.

<i>title</i>	<i>author</i>	<i>position</i>
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

*authors*

<i>title</i>	<i>keyword</i>
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

*keywords*

<i>title</i>	<i>pub_name</i>	<i>pub_branch</i>
Compilers	McGraw-Hill	New York
Networks	Oxford	London

*books4*



# 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



# 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)
```



# 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*



# 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**



# Table Inheritance

- Tables created from subtypes can further be specified as **subtables**
- 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
  - 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*
    - ▶ So we cannot create a person (tuple in *people*) who is both a student and a teacher



# Consistency Requirements for Subtables

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people*) can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers*)
  - Additional constraint in SQL:1999:

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*



# 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' ]);



# 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*)



# 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* **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*

<i>title</i>	<i>author</i>	<i>pub_name</i>	<i>pub_branch</i>	<i>keyword</i>
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  
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
```



# 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`
```



# 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



# 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 retrieved 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 directly 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 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

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



## Figure 22.05

<i>instructor</i>
<u><i>ID</i></u>
<i>name</i>
<i>first_name</i>
<i>middle_initial</i>
<i>last_name</i>
<i>address</i>
<i>street</i>
<i>street_number</i>
<i>street_name</i>
<i>apt_number</i>
<i>city</i>
<i>state</i>
<i>zip</i>
{ <i>phone_number</i> }
<i>date_of_birth</i>
<i>age</i> ( )



## Figure 22.07

