Database Systems

Lecture 10: Object-Based Databases

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Object-Based Databases

- **Complex Data Types and Object Orientation**
- Structured Data Types and Inheritance in SQL
- Array and Multiset Types in SQL
- 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
  - 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*

title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
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.

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

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

Object-Based Databases

- Complex Data Types and Object Orientation
- **Structured Data Types and Inheritance in SQL**
- Array and Multiset Types in SQL
- Persistent Programming Languages
- Comparison of Object-Oriented and Object-Relational Databases

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 and Inheritance in SQL

- 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 person of  PersonType
```

- 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 *PersonType*

  begin

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

  end

- We can now find the age of each person:

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

  from *person*

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 *teacher*s.
  - Similarly updates/deletes on *people* also result in updates/deletes on subtables
  - To override this behaviour, use “**only** *people”* in query

Object-Based Databases

- Complex Data Types and Object Orientation
- Structured Data Types and Inheritance in SQL
- **Array and Multiset Types in SQL**
- Persistent Programming Languages
- Comparison of Object-Oriented and Object-Relational Databases

Array and Multiset Types in SQL

- SQL supports two collection types:
  - Arrays
    - array types were added in SQL:1999
  - Multisets
    - multiset types were added in SQL:2003 .
- 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.
- Unlike elements in a multiset, the elements of an array are ordered
  - So we can distinguish the first item from the second item, 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;
```

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*)

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*

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 **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*

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}

Nesting

- 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

- To nest on both authors and keywords using set():

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

Nesting

- To nest on both authors and keywords using subqueries:

```
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

- To nest on both authors and keywords using subqueries:

```
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-Based Databases

- Complex Data Types and Object Orientation
- Structured Data Types and Inheritance in SQL
- Array and Multiset Types in SQL
- **Persistent Programming Languages**
- Comparison of Object-Oriented and Object-Relational Databases

Persistence of Objects

- Object-oriented programming languages already have a concept of objects
  - A type system to define object types
  - Constructs to create objects.
- These objects are transient
  - They vanish when the program terminates
  - Similar to variables in a Java or C program vanish when the program terminates.
- To turn such a language into a database programming language, the first step is to provide a way to make objects persistent.

Persistence of Objects

- Several approaches have been proposed:
  - **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-Based Databases

- Complex Data Types and Object Orientation
- Structured Data Types and Inheritance in SQL
- Array and Multiset Types in SQL
- Persistent Programming Languages
- **Comparison of Object-Oriented and Object-Relational Databases**

Object-Relational Mapping

- **Object-Relational Mapping (ORM)** systems built on top of traditional relational databases
- Implementer provides a mapping from objects to relations
- 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

Comparison of O-O and O-R Databases

- Object-relational databases are object-oriented data-bases built on top of the relation model
- Object-oriented databases are built around persistent programming languages
- Object-relational mapping systems build an object layer on top of a traditional relational database
- Each of these approaches targets a different market.

Comparison of O-O and O-R Databases

- 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.
  - SQL , however, imposes a significant performance penalty for certain kinds of applications that run primarily in main memory, and that perform a large number of accesses to the database.

Comparison of O-O and O-R Databases

- 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 they usually do not have a powerful querying capability.
  - Typical applications include CAD databases.

Comparison of O-O and O-R Databases

- Object-relational mapping systems allow programmers to build applications using an object model, while using a traditional database system to store the data.
  - They combine the robustness of widely used relational database systems, with the power of object models for writing applications.
  - However, they suffer from overheads of data conversion between the object model and the relational model used to store data.

Comparison of O-O and O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, high performance.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system

Questions?