Object-Oriented Databases

• Goal: Provide same benefits as object-oriented programming
  – Abstraction
  – Reuse
  – Natural data modeling

• A number of commercial systems
  – Gemstone (1986)
  – Informix, ObjectDB, O2, …
Object-Oriented Databases

• Often programming language model
  – No separate query language
  – “Persistent Stores”
• But this gives up many of the advantages of Relational DB
  – Query optimization
  – Efficient concurrency control (we’ll discuss later)
  – Data independence

Announcements

• Assignment 3 is out
  – Due 10/24
  – Expect Project 3 right after that
• Midterm 2 11/9
• Midterm course evaluations open to 10/21
  – https://my.cs.purdue.edu/courses
  – Your chance to provide anonymous feedback on how I can improve
Solution: Object-Relational DB

• Incorporate key features into relational model
  – User-defined data types
  – User-defined operations on those data types

• Postgres: Research project at Berkeley
  – Now available open source

• IBM bought Informix, Oracle included object-relational features
  – And almost nothing left of pure object-oriented DB
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

- Goal: Intuitive modeling of complex data
  - Abstraction
- Basic idea: Non-atomic domains
  - Cell can contain something other than “atomic” (indivisible) value
  - Example of non-atomic domain: set of integers, or set of tuples
- What part of the relational model does this violate?
  A. Everything represented as a relation (table)
  B. First normal form
  C. Relational algebra
  D. Declarative query language
- “Standardized” in SQL:1999
  - But most commercial systems vary from standard
Array and Multiset Types in SQL

- Example of array and multiset declaration:

```sql
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
  ```sql
  array ['Silberschatz', 'Korth', 'Sudarshan']
  ```

- Multisets
  ```sql
  multiset ['computer', 'database', 'SQL']
  ```

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

- To insert the preceding tuple into the relation books
  ```sql
  insert into books
  values
   ('Compilers', array ['Smith', 'Jones'],
    new Publisher ('McGraw-Hill', 'New York'),
    multiset ['parsing', 'analysis']);
  ```
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
  - Idea: Model as cells that contain relations

<table>
<thead>
<tr>
<th>title</th>
<th>author_array</th>
<th>publisher</th>
<th>keyword_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilers</td>
<td>[Smith, Jones]</td>
<td>(McGraw-Hill, NewYork)</td>
<td>[parsing, analysis]</td>
</tr>
<tr>
<td>Networks</td>
<td>[Jones, Frick]</td>
<td>(Oxford, London)</td>
<td>[Internet, Web]</td>
</tr>
</tbody>
</table>

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.
Querying Collection-Valued Attributes

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

Unnesting

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

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
<th>pub_name</th>
<th>pub_branch</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilers</td>
<td>Smith</td>
<td>McGraw-Hill</td>
<td>New York</td>
<td>parsing</td>
</tr>
<tr>
<td>Compilers</td>
<td>Jones</td>
<td>McGraw-Hill</td>
<td>New York</td>
<td>parsing</td>
</tr>
<tr>
<td>Compilers</td>
<td>Smith</td>
<td>McGraw-Hill</td>
<td>New York</td>
<td>analysis</td>
</tr>
<tr>
<td>Compilers</td>
<td>Jones</td>
<td>McGraw-Hill</td>
<td>New York</td>
<td>analysis</td>
</tr>
<tr>
<td>Networks</td>
<td>Jones</td>
<td>Oxford</td>
<td>London</td>
<td>Internet</td>
</tr>
<tr>
<td>Networks</td>
<td>Frick</td>
<td>Oxford</td>
<td>London</td>
<td>Internet</td>
</tr>
<tr>
<td>Networks</td>
<td>Jones</td>
<td>Oxford</td>
<td>London</td>
<td>Web</td>
</tr>
<tr>
<td>Networks</td>
<td>Frick</td>
<td>Oxford</td>
<td>London</td>
<td>Web</td>
</tr>
</tbody>
</table>
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`:
  ```sql
  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:
  ```sql
  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`
  ```sql
  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
  ```
Structured Types in SQL

- **Structured types** (a.k.a. *user-defined types*) can be declared and used in SQL

  ```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

  ```sql
  create table person
  (name Name,
   address Address,
   dateOfBirth date)
  ```

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

Structured Types (cont.)

- **User-defined row types**

  ```sql
  create type PersonType as
  (name Name,
   address Address,
   dateOfBirth date)
  not final
  ```

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

    ```sql
    create table customer of CustomerType
    ```

  - Alternative using **unnamed row types**.

    ```sql
    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');
    ```
What Object-Oriented feature is missing from ORDBMS?

A. Declarative queries  
B. Abstract data types  
C. Inheritance  
D. User-defined types  
E. Methods

**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.
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.
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;
  ```