Chapter 9: Object-Relational Databases

- Nested Relations
- Complex Types and Object Orientation
- Querying with Complex Types
- Creation of Complex Values and Objects
- 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.

Nested Relations

- 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.

1NF Version of Nested Relation

- 1NF version of 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>

4NF Decomposition of Nested Relation

- Remove awkwardness of flat-books by assuming that the following multivalued dependencies hold:
  - title → author
  - title → keyword
  - title → pub-name, pub-branch
- Decompose flat-doc into 4NF using the schemas:
  - (title, author)
  - (title, keyword)
  - (title, pub-name, pub-branch)

4NF Decomposition of flat-books

- 4NF design requires users to include joins in their queries.
- 1NF relational view flat-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.
Complex Types and SQL:1999

- Extensions to SQL to support complex types include:
  - 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
  - Our description is mainly based on the SQL:1999 standard
  - 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
  - We present some features that are not in SQL:1999
  - These are noted explicitly

Large Object Types

- clob: Character large objects
  - book-review clob(10KB)
- blob: Binary large objects
  - image blob(10MB)
  - movie blob (20B)
- JDBC/ODBC provide special methods to access large objects in small pieces
  - Similar to accessing operating system files
  - Application retrieves a locator for the large object and manipulates the large object from the host language

Structured and Collection Types (Cont.)

- Structured types allow composite attributes of E-R diagrams to be represented directly.
- Unnamed row types can also be used in SQL:1999 to define composite attributes
  - E.g., we can omit the declaration of type Publisher and instead use the following in declaring the type Book
    - publisher row (name varchar(20), branch varchar(20))

Structured Types (Cont.)

- We can create tables without creating an intermediate type
  - For example, the table books could also be defined as follows:
    - create table books
      - (title varchar(20),
       author-array varchar(20) array[10],
       pub-date date,
       publisher Publisher)

Creation of Values of Complex Types

- Values of structured types are created using constructor functions
  - E.g., Publisher('McGraw-Hill', 'New York')
  - Note: a value is not an object
- SQL:1999 constructor functions
  - E.g.,
    - create function Publisher(n varchar(20), a varchar(20)) returns Publisher
      - begin
        set name=n, branch=a
      - end

- Array construction
  - array ['Silberschatz', 'Korth', 'Sudarshan']
- Set value attributes (not supported in SQL:1999)
  - set v1, v2, ..., vn
- To create a tuple of the Books relation
  - (Compliance, array ['Smith', 'Jones'], Publisher('McGraw Hill', 'New York'), set 'Parsing', 'Analysis')
- To insert the preceding tuple into the relation books
  - insert into books
    - values
      - (Compliance, array ['Smith', 'Jones'],
       Publisher('McGraw Hill', 'New York'),
       set 'Parsing', 'Analysis')
Inheritance

- Suppose that we have the following type definition for people:
  ```sql
  create type Person
  (name varchar(20),
   address varchar(20))
  ```
- Using inheritance to define the student and teacher types
  ```sql
  create type Student
  under Person
  (degree varchar(20),
   department varchar(20))
  ```
  ```sql
  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

- Table inheritance allows an object to have multiple types by allowing an entity to exist in more than one table at once.
  - E.g. people table: `create table people of Person`

Table Inheritance: Consistency Requirements

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. `Person`) can correspond to at most one tuple in each of the subtables (e.g. `Student` and `Teacher`).
  - 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.

Table Inheritance: Storage Alternatives

- Storage alternatives:
  1. Store only local attributes and the primary key of the supertable in the supertable.
     - Derived attributes derived by means of a join with the supertable.
  2. Each table stores all inherited and locally defined attributes.
     - Subtables implicitly contain (inherited attributes of) all tuples in their subtables.
     - Access to all attributes of a tuple is faster: no join required.
     - If entities must have most specific type, tuple is stored only in one table, where it was created.
     - Otherwise, there could be redundancy.

Reference Types

- Object-oriented languages provide the ability to create and refer to objects.
- In SQL:1999:
  - References are to tuples, and
  - References must be scoped,
    - i.e., can only point to tuples in one specified table
  - We will study how to define references first, and later see how to use references.

Reference Declaration in SQL:1999

- E.g. 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:
  ```sql
  create type Department(
    name varchar(20),
    head ref(Person) scope people)
  ```
- We can then create a table `departments` as follows:
  ```sql
  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:
  ```sql
  create table departments of Department
  (head with options scope people)
  ```
Intializing Reference Typed Values

- In Oracle, to create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately by using the function ref(p) applied to a tuple variable.
- E.g. to create a department with name CS and head being the person named John, we use:
  ```sql
  insert into departments
  values ('CS', null)
  update departments
  set head = (select ref(p) from people as p
               where name = 'John')
  where name = 'CS'
  ```

User Generated Identifiers

- SQL:1999 allows object identifiers to be user-generated:
  - 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.
- E.g.
  ```sql
  create type Person
  (name varchar(20)
   address varchar(20))
  ref using varchar(20)
  create table people of Person
  ref is oid user generated
  ```

- When creating a tuple, we must provide a unique value for the identifier (assumed to be the first attribute):
  ```sql
  insert into people values
  ('01234567', 'John', '23 Coyote Run')
  ```

Path Expressions

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

Collection-Value Attributes

- Collection-valued attributes can be treated much like relations, using the keyword unnest:
  - The books relation has array-valued attribute author-array and set-valued attribute keyword-set.
- To find all books that have the word "database" as one of their keywords,
  ```sql
  select title
  from books
  where 'database' in (unnest(keyword-set))
  ```
- Note: Above syntax is valid in SQL:1989, but the only collection type supported by SQL:1999 is the array type.
- To get a relation containing pairs of the form "title, author-name" for each book and each author of the book:
  ```sql
  select title, A
  from books as B, unnest (B.author-array) as A
  ```

Collection Valued Attributes (Cont.)

- We can access individual elements of an array by using indices:
  ```sql
  select author-array[1], author-array[2], author-array[3]
  from books
  where title = 'Database System Concepts'
  ```
Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called unnesting.
- E.g. 
  ```sql
  select title, A as author, publisher.name as pub_name,
  publisher.branch as pub_branch, K as keyword
  from books as B, unnest(B.author-array) as A, unnest (B.keyword-list) as K
  ```

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the select clause.
  ```sql
  select title,
  ( select author
  from flat-books as M
  where M.title=O.title ) as author-set,
  publisher.name as pub_name, publisher.branch as pub_branch
  from flat-books as O
  ```

SQL Functions

- Define a function that, given a book title, returns the count of the number of authors (on the 4NF schema with relations books4 and authors).
  ```sql
  create function author-count(title varchar(20))
  returns integer
  begin
    declare a-count integer;
    select count(*) into a-count
    from authors
    where authors.title = title
  end
  ```

Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - E.g. functions to check if polygons overlap, or to compare images for similarity
  - Some databases support table-valued functions, which can return a relation as a result
  - SQL:1999 also supports a rich set of imperative constructs, including
    - Loops, if-then-else, assignment
  - Many databases have proprietary procedural extensions not in SQL that differ from SQL:1999

SQL Methods

- Methods can be viewed as functions associated with structured types
  - They have an implicit first parameter called self which is set to the structured-type value on which the method is invoked
  - The method code can refer to attributes of the structured-type value using the self variable
    - E.g. self.a

External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions
  ```sql
  create procedure author-count-proc(title varchar(20),
  out count integer)
  language C
  external name '/usr/avi/bin/author-count Proc'
  ```

SQL Functions and Procedures (cont.)

- The author-count function could instead be written as procedure:
  ```sql
  create procedure author-count-proc (in title varchar(20),
  out a-count integer)
  begin
    select count(*) into a-count
    from authors
    where authors.title = title
  end
  ```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the call statement.
  - E.g. from an SQL procedure
    ```sql
    declare a-count integer;
    call author-count-proc('Database systems Concepts', a-count);
    ```
  - SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as the number of arguments differ, or at least the types of the arguments differ.
Finding all employees of a manager

- Procedure to find all employees who work directly or indirectly for mgr
- Relation manager(empname, mgrname) specifies who directly works for whom
- Result is stored in emp(name)

create procedure findEmp(mgr char(10))
begin
create temporary table newemp(name char(10));
create temporary table temp(name char(10));
insert into newemp
    select empname
    from manager
    where mgrname = mgr;
end;
A Partially Nested Version of the \textit{flat-books} Relation

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
<th>publisher</th>
<th>keyword-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complier</td>
<td>Smith</td>
<td>(McGraw-Hill, New York)</td>
<td>(programming, analysis)</td>
</tr>
<tr>
<td>Compiler</td>
<td>Jones</td>
<td>(McGraw-Hill, New York)</td>
<td>(programming, analysis)</td>
</tr>
<tr>
<td>Networks</td>
<td>Jones</td>
<td>(Oxford, London)</td>
<td>(Internet, Web)</td>
</tr>
<tr>
<td>Networks</td>
<td>Pick</td>
<td>(Oxford, London)</td>
<td>(Internet, Web)</td>
</tr>
</tbody>
</table>

Database System Concepts 9.50 ©Silberschatz, Korth and Sudarshan