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 $\equiv$ indivisible)
  - Example of non-atomic domain: set of integers, or set of tuples
  - Allows more intuitive modelling for applications with complex data
- Intuitive definition:
  - allow relations wherever we allow atomic (scalar) values — relations within relations
- Retains mathematical foundation of relational model
- Violates first normal form

Example of a Nested Relation

- Example: document retrieval system
- Each document has
  - title,
  - a set of authors,
  - date acquired, and
  - a set of keywords
- Non-1NF document relation

title	*author-list*	*date*	*keyword-list*
		day month year	
salesplan	{Smith, Jones}	1 April 79	{profit, strategy}
status report	{Jones, Frick}	17 June 85	{profit, personnel}

doc

1NF Version of Nested Relation

- 1NF version of *doc*

title	*author*	*day*	*month*	*year*	*keyword*
salesplan	Smith	1	April	79	profit
salesplan	Jones	1	April	79	profit
salesplan	Smith	1	April	79	strategy
salesplan	Jones	1	April	79	strategy
status report	Jones	17	June	85	profit
status report	Frick	17	June	85	profit
status report	Jones	17	June	85	personnel
status report	Frick	17	June	85	personnel

flat-doc

4NF Decomposition of Nested Relation

- Remove awkwardness of *flat-doc* by assuming that the following multivalued dependencies hold:
  - $title \rightarrow\rightarrow author$
  - $title \rightarrow\rightarrow keyword$
  - $title \rightarrow day\ month\ year$
- Decompose *flat-doc* into 4NF using the schemas:
  - (*title, author*)
  - (*title, keyword*)
  - (*title, day, month, year*)

4NF Decomposition of *flat – doc*

title	*author*
salesplan	Smith
salesplan	Jones
status report	Jones
status report	Frick

title	*keyword*
salesplan	profit
salesplan	strategy
status report	profit
status report	personnel

title	*day*	*month*	*year*
salesplan	1	April	89
status report	17	June	94

Problems with the 4NF Schema

- 4NF design requires users to include joins in their queries.
- 1NF relational view *flat-doc* 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.
- Nested relation representation is much more natural here

Complex Types and Object Orientation

- Extensions to relational model include:
  - Nested relations
  - Complex types
  - Specialization (**is-a** hierarchies)
  - Inheritance
  - Object identity
- Will cover SQL extensions
  - SQL-3 standards currently (as of early 1997) being developed
  - Our presentation is based loosely on an SQL-3 draft, the XSQL language and the Illustra extensions to SQL

Structured and Collection Types

Define new types and a new table

- **create type** *MyString* **char varying**.
- **create type** *MyDate*
  (*day* **integer**,
  month **char(10)**,
  year **integer**)
- **create type** *Document*
  (*name MyString*,
  author-list **setof**(*MyString*),
  date MyDate,
  keyword-list **setof**(*MyString*))
- **create table** *doc* **of type** *Document*

Structured and Collection Types (Cont.)

- Unlike table definitions in ordinary relational databases, the *doc* table definition allows attributes that are sets and structured attributes like *MyDate*.
- Allows composite attributes and multivalued attributes of E-R diagrams to be represented directly.
- The types created using the above statements are recorded in the schema stored in the database.
- Can create tables directly.

```
create table doc
      (name MyString,
       author-list setof(MyString),
       date MyDate,
       keyword-list setof(MyString))
```

Inheritance of Types

- Consider the following type definition for people.

```
create type Person
     (name MyString,
      social-security integer)
```

- Use inheritance to define student and teacher types.

```
create type Student
     (degree MyString,
      department MyString)
     under Person
create type Teacher
     (salary integer,
      department MyString)
     under Person
```

Inheritance of Types (Cont.)

- To store information about teaching assistants and to avoid a conflict between two occurrences of *department*, use an **as** clause.
- Definition of the type *TeachingAssistant*.

```
create type TeachingAssistant
        under Student with (department as student-dept),
                Teacher with (department as teacher-dept)
```

Inheritance at the Level of Tables

- Allows an object to have multiple types by allowing an entity to exist in more than one table at once.
- *people* table:

```
create table people
        (name MyString,
         social-security integer)
```

- Can then define the *students* and *teachers* tables as follows.

```
create table students
        (degree MyString,
         department MyString)
        under people
create table teachers
        (salary integer,
         department MyString)
        under people
```

Table Inheritance: Roles

- Table inheritance is useful for modelling *roles*
  - permits an object to have multiple types, without having a *most-specific* type (unlike type inheritance).
  - e.g., an object can be in the *students* and *teachers* subtables simultaneously, without having to be in a subtable *student-teachers* that is under both *students* and *teachers*
  - object can gain/lose roles: corresponds to inserting/deleting object from a subtable

Table Inheritance: Consistency Requirements

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable *people* can correspond to at most one tuple of each of the tables *students* and *teachers*.
  - Each tuple in *students* and *teachers* must have exactly one corresponding tuple in *people*.
- Inherited attributes other than the primary key of the supertable need not be stored, and can be derived by means of a join with the supertable, based on the primary key.
- As with types, multiple inheritance is possible.

Reference Types

- Object-oriented languages provide the ability to create and refer to objects.
- Redefine the author-list field of the type *Document* as:

  author-list **setof**(**ref**(*Person*))

  Now *author-list* is a set of references to *Person* objects
- Tuples of a table can also have references to them.
  - References to tuples of the table *people* have the type **ref**(*people*).
  - Can be implemented using either primary keys or system generated tuple identifiers.

Relation Valued Attributes

- By allowing an expression evaluating to a relation to appear anywhere a relation name may appear, our extended SQL can take advantage of the structure of nested relations.
- Consider the following relation *pdoc*.

create table *pdoc*
    name MyString,
    author-list **setof**(**ref**(*people*)),
    date MyDate,
    keyword-list **setof**(*MyString*))

Example Queries

- Find all documents which have the word “database” as one of their keywords.

  select *name*
  from *pdoc*
  where “database” **in** *keyword-list*

- Create a relation containing pairs of the form “document-name, author-name” for each document and each author of the document.

  select *B.name*, *Y.name*
  from *pdoc* **as** *B*, *B.author-list* **as** *Y*

- Find the name, and the number of authors for each document.

  select *name*, **count**(*author-list*)
  from *pdoc*

Path Expressions

- The dot notation for referring to composite attributes can be used with references.

- Consider the previous table *people* and a table *phd-student*.

  create table *phd-students*
      (*advisor* **ref**(*people*))
      under *people*

- Find the names of the advisors of all Ph.D. students.

  select *students.advisor.name*
  from *phd-students*

- Find the names of all authors of documents in the *pdoc* relation.

  select *Y.name*
  from *pdoc.author-list* **as** *Y*

Unnesting

- Transformation of a nested relation into first normal form.
- Converts a nested relation into a single flat relation with no nested relations or structured types as attributes.
- Unnest the *doc* relation (*author-list* and *keyword-list* are nested relations; *name* and *date* are not nested).

```
select name, A as author, date.day, date.month, date.year,
            K as keyword
from doc as B, B.author-list as A, B.keyword-list as K
```

- *B* in the from clause is declared to range over *doc*.
- *A* ranges over the authors in *author-list* for that document
- *K* is declared to range over the keywords in the *keyword-list* of the document.

Nesting

- Transforming a 1NF relation into a nested relation.
- Can be carried out by an extension of grouping in SQL.
- Nest the relation *flat-doc* on the attribute *keyword*:

select *title*, *author*, (*day, month, year*) **as** *date*,
        set(*keyword*) **as** *keyword-list*
from *flat-doc*
groupby *title*, *author*, *date*

title	*author*	*date*	*keyword-list*
		(*day, month, year*)	
salesplan	Smith	(1, April, 89)	{profit, strategy}
salesplan	Jones	(1, April, 89)	{profit, strategy}
status report	Jones	(17, June, 94)	{profit, personnel}
status report	Frick	(17, June, 94)	{profit, personnel}

Functions

- Define a function that, given a document, returns the count of the number of authors.

```
create function author-count(one-doc Document)
        returns integer as
        select count(author-list)
        from one-doc
```

- Find the names of all documents that have more than one author.

```
select name
from doc
where author-count(doc) > 1
```

Functions (Cont.)

Database system may also allow the use of functions written in other languages such as C or C++

- Benefits: more efficient for many operations, more expressive power
- Drawbacks
  - code to implement function may need to be loaded into database system
  - risk of accidental corruption of database structures
  - security risk

Creation of Complex Values and Objects

- Create a tuple of the type defined by the *doc* relation

  (“salesplan”, **set**(“Smith”, “Jones”), (1, “April”, 89),
  set(“profit”, “strategy”))

  - value for the composite attribute *date* is created by listing its attributes day, month and year within parentheses.
  - set valued attributes *author-list* and *keyword-list* are created by enumerating their elements within parentheses following the keyword **set**.

Example Queries

- Insert the above tuple into the relation *doc*.

  insert into doc
  values
  (“salesplan”, **set**(“Smith”, “Jones”), (1, “April”, 89),
  set(“profit”, “strategy”))

- Can use complex values in queries. Find the names and dates of all documents whose name is one of “salesplan”, “opportunities” or “risks”.

  select *name*, *date*
  from *doc*
  where *name* **in set**(“salesplan”, “opportunities”, “risks”)

Additional Concepts

- Multiset values can be created by replacing **set** by **multiset**.
- Use *constructor* functions to create new objects.
  - constructor function for an object of type T is $T()$
  - creates a new uninitialized object of type T, fills in its **oid** field, and returns the object
  - fields of the object must then be initialized

Comparison of O-O and O-R Databases

Summary of strengths of various database systems:

- Relational systems: simple data types, querying, high protection.
- Persistent programming language based OODBs: complex data types, integration with programming language, high performance.
- Object-relational systems: complex data types, querying, high protection.