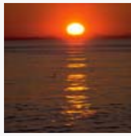
Chapter 9: Object-Based Databases

Database System Concepts



Chapter 9: 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 ≡ 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 set of authors,
  - Publisher, and
  - a set of keywords
- Non-1NF relation *books*

title	*author-set*	*publisher*	*keyword-set*
		(*name*, *branch*)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

4NF Decomposition of Nested Relation

- Remove awkwardness of *flat-books* by assuming that the following multivalued dependencies hold:
  - *title* $\twoheadrightarrow$ *author*
  - *title* $\twoheadrightarrow$ *keyword*
  - *title* $\twoheadrightarrow$ *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*

title	*author*
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

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

Problems with 4NF Schema

- 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

Structured Types and Inheritance in SQL

- Structured types can be declared and used in SQL

  create type *Name* **as**
  (first*name* **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 *customer* (
  name Name,
  address Address,
  dateOfBirth **date**)
- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- User-defined row types

  create type *CustomerType* **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*

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*

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 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*)

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
```

- Similar to the nested relation books, but with array of authors instead of set

Creation of Collection Values

- Array construction

  array [‘Silberschatz’,`Korth’,`Sudarshan’]
- Multisets
  - **multisetset** [‘computer’, ‘database’, ‘SQL’]
- To create a tuple of the type defined by the *books* relation:

  (‘Compilers’, **array**[`Smith’,`Jones’],
  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’],
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 us 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*)

Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- NOTE: SQL:1999 does not support nesting
- 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*

- 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*

1NF Version of Nested Relation

1NF version of *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

flat-books

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the **select** clause.

```
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*)

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:
  - by class - explicit declaration of persistence
  - by creation - special syntax to create persistent objects
  - by marking - make objects persistent after creation
  - 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)

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.
- 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

Database System Concepts