Chapter 9

Object-Relational Databases

1. The object-relational data model extends the relational model by providing rich type system including object orientation and add constructs to relational query languages to deal with added (complex) data types.

2. Such extensions attempt to preserve the relational foundations — in particular, the declarative access to data — while extending the modeling power.

9.1 Nested Relations

1. 1NF requires that all attributes have atomic (indivisible) domains.

2. The nested relational model is an extension of the relational model in which domains may be either atomic or relation-valued. This allows a complex object to be represented by a single tuple of a nested relation — one-to-one correspondence between data items and objects.

3. Suppose the information to be stored consists of (i) document title, (ii) author_list (set of authors), (iii) date (day, month, year), and (iv) key_word_list (list of key words).

4. Example: A non-1NF document relation, *doc*.

title	author_list	date	keyword_list
		day month year	
salesplan	{Smith, Jones}	1 April 89	{profit, strategy}
stat. report	{Jones, Frick}	17 July 94	{profit, personnel}

5. The *doc* relation can be represented in 1NF, *doc'*, but awkward. If we assume the following multi-value dependencies (MVDs) hold:

   ◇ title $\twoheadrightarrow$ author
   ◇ title $\twoheadrightarrow$ keyword
   ◇ title $\rightarrow$ day month year

   we can decompose the relation into 4NF using the schemes:
   (title, author)
   (title, keyword)
   (title, day, month, year)

6. But the non-1NF representation may be an easier-to-understand model (closer to user's view). The 4NF design would require users to include joins in their queries, thereby complicating interaction with the system. We could define a view, but we lose the one-to-one correspondence between tuples and documents.

9.2 Complex Types and Object Orientation

1. The extensions include nested relations, nested records, inheritance, references to objects, and other object-oriented features.

2. The presentation is based on a draft of SQL-3, with the incorporation of features from XSQL and Illustra.

9.2.1 Structured and collection types

1. Define a relation *doc* with complex attributes: sets and structured attributes.

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

2. It allows type definition recorded in the schema stored in the database. One can also create table directly, without creating an intermediate type for the table.

3. Complex type systems usually support other collection types, such as arrays and multisets, e.g.,

```
author-array MyString[10] // presents an ordered list of authors.
print-runs multiset(integer) // presents the number of copies in each printing run.
```

9.2.2 Inheritance

1. Inheritance can be at the level of types or at the level of tables.

2. Inheritance of types.

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

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

3. Multiple inheritance: Since *name* and *social-security* are inherited from a common source *Person*, there is no conflict by inheriting them from *Student* as well as *Teacher*. However, *department* is defined separately in *Student* and *Teacher*, and one can rename them to avoid a conflict.

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

4. Inheritance of tables:

   To avoid creation of too many subtypes, one approach in the context of database systems is to allow an object to have multiple types without having a most specific type.

   Object-relational systems can model such a feature by using inheritance at the level of tables, rather than types, and allowing an entity to exist in more than one table at once.

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

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

5. There are consistency requirements on subtables and supertables.

   - Each tuple of supertable *people* can correspond to (i.e., having the same values for all inherited attributes as) at most one tuple in each of the tables *students* and *teachers*. (Why?)
   - Each tuple in *students* and *teachers* must have exactly one corresponding tuple in *people*. (Why?)

6. Subtables can be stored in an efficient manner without replication of all inherited fields. Inherited attributes other than the primary key of the supertables can be derived by means of a join with the supertable, based on the primary key.

7. Multiple inheritance is possible with tables. A teaching-assistant can simply belong to the table *students* as well as to the table *teacher*. However, if we want, we can create a table for teaching-assistant entities as follows:

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

   Based on the consistency requirements for subtables, if an entity is present in the *teaching-assistant* table, it is also present in the *teachers* and in the *students* table.

8. Inheritance makes schema definition natural, ensures referential and cardinality constraints, enables the use of functions defined for supertypes on object belonging to subtypes, and allows the orderly extension of a database system to incorporate new types.

9.2.3 Reference Types

1. Reference to objects: An attribute of a type can be a reference to an object of a specific type, e.g., **ref**(*Person*).

2. Reference to tuples: Tuples of a table can also have references to them. E.g., **ref**(*people*). It can be implemented using the primary key or tuple-id.

3. SQL-3 uses **identity** (for tuples) and **oid** for objects.

9.3 Querying with Complex Types

9.3.1 Relation-Valued Attributes

1. Our extended SQL allows an expression evaluating to a relation to appear anywhere that a relation name may appear. The ability to use subexpressions freely makes it possible to take advantage of the structure of nested relations.

2. The schema for a relation *pdoc*.

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

3. To find all documents having a keyword "database",

   select *name*
   from *pdoc*
   where *"database"* **in** *keyword-list*

4. To find pairs of the form "doc-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*

5. Aggregate functions can be applied to any relation-valued expression.

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

9.3.2 Path Expressions

1. An expression of the form "*student.advisor.name*" is called a *path-expression*. References can be used to hide join operations and thus the use of references simplifies the query considerably.

2. Example.

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

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

3. In general, attributes used in a path expression can be a collection, such as a set or a multiset. E.g., to get names of all authors of documents, we have

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

9.3.3 Nesting and Unnesting

1. The transformation of a nested relation into 1NF is called *unnesting*.

2. Example. To complete unnest the doc relation, we have

   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*

3. The reverse operation of transformation of a 1NF relation into a nested relation is called *nesting*.

4. Example. To nest the relation *flat-doc* on the attribute *keyword*, we have

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

   This wil generate the following table.

title	author_list	date	keyword_list
		day month year	
salesplan	Smith	1 April 89	{profit, strategy}
salesplan	Jones	1 April 89	{profit, strategy}
status report	Jones	17 July 94	{profit, personnel}
status report	Frick	17 July 94	{profit, personnel}

5. Example. To convert *flat-doc* back to the nested table *doc*, we have

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

9.3.4 Functions

1. Object-relational systems allow functions to be defined by users.

2. Functions can be defined in a data manipulation language such as extended SQL.

   Example. Given a document, return the count of the number of authors.

   create function *author-count(one-doc Document)*
   return integer as
   select count*(author-list)*
   from *one-doc*

   The function can be used in a query,

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

   Note that, although *doc* refers to a relation in the **from**-clause, it is treated as a tuple variable in the **where**-clause, and can therefore be used as an argument to the *author-count* function.

3. In general, a select statement can return a collection of values. If the return type of a function is a collection type, the result of the function is the entire collection. However, if the return type is not a collection type, the collection generated by SQL should contain only one tuple. Otherwise, a system may have two choices: flag an error or select an arbitrary one from the collection.

4. Functions can also be defined in a programming language such as C or C++. It can be more efficient and handle more complex computations than that defined using SQL.

   However, since the code needs to be loaded and executed with the database system code, it may carry the risk of

   (a) integrity: a bug in the program can corrupt the database internal structures, and

   (b) security: it can by-pass the access control functionality of the database system.

5. Embedded SQL is different from C++ code functions: In embedded SQL, the query is passed by the user program to the database system to be run. User-written code never needs to access to the database itself. The operating system thus can protect the database from access by any user process.

9.4 Creation of Complex Values and Objects

1. Create and updates tuples with complex types (tuple-valued and set-valued)

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

2. We can also use complex values in queries: anywhere in a query where a set is expected, we can enumerate a set.

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

3. To create new objects, we can use *constructor* functions. The *constructor* function for an object type T is $T()$; when it is invoked it creates a new uninitialized object of type T, fills in its **oid** field, and returns the object. The fields of the object must then be initialized.

9.5 Comparison of Object-Oriented and Object-Relational Databases

1. Object-relational databases are object-oriented databases built on top of the relational model.

   The declarative nature and limited power of the SQL language provides good protection of data from programming errors, and makes the high-level optimization, such as reducing I/O, relatively easy.

   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.

2. Persistent programming language-based OODBs target applications of that form that have high performance requirements: no data translation needed, low-overhead access to persistent data, but more susceptible to data corruption to programming errors and usually do not have a powerful querying capability. Typical applications include CAD databases.

   This contrasts with a declarative language which 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 databases.

3. Summary:

   (a) **relational systems:** simple data types, powerful query languages, high protection.

   (b) **persistent programming language-based OODBs:** complex data types, integration with programming languages, high performance.

   (c) **object-relational systems:** complex data types, powerful query languages, high protection.

   Some systems blurs the boundary, e.g., some object-oriented database systems built around a persistent programming language are implemented on top of a relational database system.

9.6 References

1. A. Silberschatz, H. F. Korth and S. Sudarshan, *Database System Concepts*, McGraw-Hill, 3rd ed., 1997.