SIMPLIFYING QUERIES AND UPDATES WITH INHERITANCE
How Polyhedra's object-oriented features allow a more rational database design than is possible with a purely relational DBMS.

Abstract

Traditional relational database management systems are simple to understand, but it can be difficult to model 'real-world' systems – especially if you want to denormalise the database in the interests of data integrity. The Polyhedra Relational DBMS products allow tables to be derived from other tables, making it easier for object-oriented applications to store and retrieve the information they need, which has the nice side-effect of improving both performance and overall integrity of the data and the applications that use it.

Introduction: why there is a problem

The object-oriented paradigm has achieved great popularity in software project over the last two decades, as witnessed by the explosive growth in the use of the C++, Java and C# programming languages. The OO approach allows not just for a nice way of encapsulating the data structure in an application with the functions ("methods") that apply to them, but it also allows the natural relationship between types of object to be modelled and exploited.

When it comes to database systems, though, the ruling paradigm is that or the relational database. There are good reasons for this, not least of which is simplicity: a relational database is simply a collection of named tables with each column in a table having a name and a type. As almost all relational DBMS's use SQL as the interface language and most have libraries that support the ODBC and JDBC APIs, developers are likely to be familiar with them, and thus adoption costs are low. Also, as relational database management system will make sure that information about the 'schema' (the metadata about the table structures) is viewable, it is possible to write general-purpose reporting tools.

When used together, though, problematic. Consider the case where an application has a class A, with attributes a1, a2 and a3 along with a numeric field called id that uniquely identifies members of the class. Now add a derived class B, with additional attributes b1, b2 and b3. Each 'B' type object has actually 7 attributes: id, a1, a2, a3, b1, b2, b3. The 'standard' way of storing such information in a relational database would be to use two tables, linked by a foreign key:

ENEA

```
create table A
( id integer primary key
, a1 large varchar
, a2 large varchar
, a3 large varchar
);

create table B
( id integer primary key references A
, b1 large varchar
, b2 large varchar
, b3 large varchar
);
```

To see some of the problems that arise, let us consider the SQL that would be needed to insert, update, query and delete a record that holds information about a 'B' object:

```
-- insertion:
insert into A values (1, 'a1 value', 'a2 value', 'a3 value');
insert into B values (1, 'b1 value', 'b2 value', 'b3 value'); commit;

-- update three fields:
update A set a1='new a1 value' where id=1;
update B set b2='new b2 value', b3='new b3 value' where id=1; commit;

-- query (using SQL92 syntax for inner joins):
select a.id, a.a1, a.a2, a.a3, b.b1, b.b2, b.b3 from a,b where b.id=a.id and a.id=1;

-- deletion (assuming 'cascaded delete' is not available, or not desired):
delete from B where id=1; delete from A where id=1; commit;
```

As you can see, everything involves explicitly accessing both the A and B tables. If the application wants to use multiple levels of inheritance, the SQL will get progressively more complicated. Some of this can be hidden using views and stored procedures, but this merely disguises the underlying situation.

Table inheritance in Polyhedra

The Polyhedra product line was designed from the start for use with applications that used object-oriented techniques, and has an inbuilt mechanism to represent class hierarchies. It is very simple to use, as the syntax of the CREATE TABLE command has been extended to include a 'DERIVED FROM' clause. Thus, looking at our earlier example, you can define tables A and B as follows:

```
create table A
( id integer primary key
, a1 large varchar
, a2 large varchar
, a3 large varchar
);

create table B
( derived from A
, b1 large varchar
, b2 large varchar
, b3 large varchar
);
```

If you create a record in A, it will have 4 fields, as one would expect – but a record created in B will have a total of 7 fields: the four inherited from A, plus the 3 new fields declared in the definition of table B. To see the advantages of this approach, let us look again at the SQL that would be needed to insert, update, query and delete a record that holds information about a 'B' object:

```
-- insertion:
insert into B values (1, 'a1 value', 'a2 value', 'a3 value',
                         'b1 value', 'b2 value', 'b3 value'); commit;

-- update three fields:
update B set a1='new a1 value', b2='new b2 value', b3='new b3 value' where id=1;
commit;

-- query:
select id, a1, a2, a3, b1, b2, b3 from b where id=1;

-- deletion:
delete from B where id=1; commit;
```

This approach makes everything more 'natural' and simpler for the application writer, which means code is more understandable and errors are less likely. You should note that in proper object-oriented style each object created in B can also be considered an object of type A; not only will the records magically appear in that table, but can also be operated on via that table:

```
-- insertion:
insert into B values (1, 'a1 value', 'a2 value', 'a3 value',
                         'b1 value', 'b2 value', 'b3 value'); commit;

-- update two fields:
update A set a1='another a1 value', a2='new a2 value' where id=1; commit;

-- query:
select id, a1, a2 from a where id=1;

-- deletion:
delete from A where id=1; commit;
```

The original versions of Polyhedra in-memory database actually stored records in inherited tables in multiple parts, according to the level of inheritance. Release 3.0 (which came out in 1997) moved to the more logical and space-efficient approach whereby the data for a record in a derived table is stored in a single data structure.

A further extension introduced by Polyhedra is the ability to flag attributes of a table as SHARED or VIRTUAL. A shared attribute is one whose value is common to all records in that table (and derived tables); in fact, the value is not stored in individual records, but instead is recorded separately with the metadata describing the table. The concept is similar to that of static values in a C++ class definition. VIRTUAL attributes are similar to STATIC attributes, but derived tables will have a different value. A common way of using this would be to have a virtual attribute called 'TYPE' in a base table, and when inserting records in derived table the type field would be set to the name of the table in which the record was inserted. It is then possible to 'look up' a record in the base table to see its type, and thus know what table to query to see the full information about an object.

The final enhancement offered by Polyhedra is the ability to add code to tables that is triggered when records are created or deleted, or when attributes are changed. Trigger code that has been defined for one table is inherited by derived tables, and it is even possible to override these 'methods', making the database server a full object-oriented computing environment with inbuilt persistent storage and external access!

Summary

Using Polyhedra's table inheritance mechanism can allow a more natural representation of the objects in a data model than can be achieved by using the standard relational model, and this leads to simplifications in the way client applications create, access and manipulate data held in the database system. This in turn can speed up development of complex applications, and make them more understandable and easier to maintain.

ENEA