Surviving Client/Server: Object Oriented Databases

by Steve Troxell

Object oriented databases have been around in one form or another for about 20 years. Early products didn't catch on, partly because they were difficult to use in comparison to traditional database management systems, and largely because object oriented techniques were not well understood at the time by developers and data modelers.

Only in the last few years have ODBMS products seemed to become serious contenders in the database market. This is due in part to a convergence in the maturity of both the products and the developers who must work with them. The products are becoming the robust development platforms required by development teams, and the developer community has become so saturated with object orientation it no longer takes a great mental leap to accept the paradigm.

An Object Oriented What?

What exactly is an object oriented database? The simplest way to begin is by anchoring ourselves in object oriented programming. As Delphi developers, the basic concepts of OOP should be familiar to us all. An object represents some entity within our application, such as a screen component or an employee. We define a class describing the attributes and behavior of the object. In Delphi classes, object attributes are defined as properties and object behaviors are defined as methods. For example, an employee class might have properties for name, date of hire, salary, and so on. We might further define employee methods like transferring to a new department, receiving a paycheck, or even being terminated.

Well-defined Delphi classes can be reused through inheritance. We would define more general characteristics in base classes and make descendants from them, adding more and more specialized functionality for specialized purposes, without having to recode the characteristics held in common. For example, we might define a `TPerson` class with `Name` and `DateOfHire` properties. We then might make descendants for `TEmployee` and `TContractor`, each inheriting all the characteristics of a `TPerson` and implementing their own additional characteristics.

Classes can also implement custom behavior based on their datatype. For example, in the `TPerson` class we could add a `Salary` method. The `TEmployee` and `TContractor` subclasses could each implement a completely different calculation for determining salary. When we call for the salary of a `TPerson`, we get answers calculated correctly without even knowing if we are working with a `TEmployee` or a `TContractor`.

We are accustomed to working with objects, or even collections of objects, in our Delphi applications. But as soon as our application shuts down, the data in these objects disappears. Object databases grant persistence to the objects by storing them in a database management system. The object instances are then sharable between different programs and different users. In addition, they are protected with all the robust handling of concurrency, transaction processing, and durability we've come to expect from a DBMS.

Think of it this way: in the structured programming days before OOP, persistence could be achieved by storing records in a simple random-access disk file. When relational database systems came along, they expanded the persistence of structured programming techniques with multi-user concurrency control, isolation, transaction processing, simplified querying and reporting capability and more.

In essence, object database systems are to object oriented programming what relational database systems were to the simple random-access file techniques of structured programming.

ODBMS Versus RDBMS

The next big question is 'How does an object oriented database differ from the relational database model I'm accustomed to?' Many of the basic concepts are the same between the two. Object databases define classes to take the place of relational tables. Classes can contain many instances, which would be the same as a table's rows. The class's properties are analogous to a table's columns. While it helps to make this comparison in terminology, it is important to remember the structures aren't *exactly* the same.

Object Identifiers (OIDs)

Relational tables are designed according to set theory. That is, the physical position of the rows is meaningless, only the fact that the data in certain rows fit a certain search criteria is relevant. The physical position of the row is not accessible in any fashion and, in fact, the physical position of the row could even change as data is manipulated in the table. Rows themselves are looked up based on searching for specific values in the actual row data. Relational databases are optimized to perform these searches very quickly, but the ability to easily navigate among rows has been lost.

Object databases assign an object identifier (OID) to each

InventNum	Category	Price	Description	ScreenSize	ClockSpeed	Capacity
10001	MON	399.99	Sony SVGA	15	(null)	(null)
10021	HDD	299.99	Maxtor	(null)	(null)	4400
10075	CPU	145.00	Intel Pent II	(null)	200	(null)
10080	MON	599.95	Gateway SVGA	17	(null)	(null)
10211	CPU	115.00	AMD K6	(null)	200	(null)
10301	HDD	199.99	Maxtor	(null)	(null)	3686
10313	HDD	1199.99	Seagate	(null)	(null)	9216

➤ *Figure 1: Items*

instance. In database terms, this is similar to an auto-incrementing field, a unique identifier automatically generated by the system. However, its value is not normally accessible by the program. An object's OID is guaranteed to remain constant throughout the lifetime of the object. To change the OID you must destroy the original object and create a new one.

Unlike an auto-incrementing field, the OID is a navigational anchor that the ODBMS can use to quickly locate and retrieve the object instance. In this respect, the OID is more closely related to a pointer to a Delphi object. Just like the Delphi program can quickly access the Delphi object in memory using the object pointer, the ODBMS can quickly access the database object using the OID. When relationships are made between objects, the OIDs are stored. In a relational model, foreign key fields would store values that identify the related record. The RDBMS would then use those values to search the related table for the matching row. With OIDs, the related object can be fetched directly.

Consider a simple invoice report. An invoice might list an order number, customer information, order date, ship date and a line for each item ordered, showing the item number and description, quantity, price and extended price. With a relational database, invoices are typically implemented with at least two tables: a header table containing one row per order with all the header information such as order number, customer and so on, and a detail table containing one row for each item ordered. The order number would be stored in each detail row, and we would fetch them by searching for rows containing a matching order number.

In a Delphi application, a `TInvoice` object would contain properties for all the header information, and might contain an `Items` property which was nothing more than a collection of pointers to `TInvoiceItem` objects describing each item in the invoice. The invoice is represented in the program by a single instance of `TInvoice`. Our `TInvoice` object is said to be a 'composite object' because it contains other object instances, namely all the `TInvoiceItems` for a given order. These objects are not physically contained in `TInvoice`. Rather, `TInvoice` holds references to them in the form of object pointers and they are normally accessed via the `TInvoice` class.

In an object database, we would have `Invoice` and `InvoiceItem` classes, storing information in much the same way as our relational tables. However, each `Invoice` object would link to `InvoiceItem` by storing within itself a list of the OIDs from `InvoiceItem`. When we retrieve an invoice, the line items are accessed more quickly because we have direct links to them, rather than having to search for objects in `InvoiceItem` with a matching order number.

The concept of the OID is a significant factor in how object oriented databases can outperform relational databases in certain cases. The object relationships are explicitly recorded as lists of OIDs independent of the data in the object. In relational databases, relationships are based on values in the data itself and must be looked up each time. As relationships become more complex, requiring more joined tables, you can begin to see how advantageous direct association via the OID can be.

Object databases can significantly out-perform their relational cousins when complex data is involved. Calculations or processes involving a number of joined tables can be real performance killers in a relational database. But an object database, with its direct relationships via the OID, can combine these results much more quickly.

Inheritance

Another major element of object-orientation is inheritance. We can extend the capabilities of a class by subclassing it, inheriting all the base class characteristics, and adding our own.

Suppose we are designing a system to manage the inventory for a computer retail store. We'll need to keep track of each item in stock: monitors, CPUs, printers, keyboards and so on. Obviously each item will have some shared characteristics such as an inventory number, price, manufacturer, and shipping weight. Some items have unique characteristics: monitors have a screen size and resolution, CPUs have a clock speed, and hard disks have a storage capacity.

To organize this data in a relational database, we have three general choices: all-in-one, independent entities, or master-detail.

Monitors

InventNum	Price	Description	ScreenSize
10001	399.99	Sony SVGA	15
10080	599.95	Gateway SVGA	17

CPUs

InventNum	Price	Description	ClockSpeed
10075	145.00	Intel Pent II	200
10211	115.00	AMD K6	200

HardDrives

InventNum	Price	Description	Capacity
10021	299.99	Maxtor	4400
10301	199.99	Maxtor	3686
10313	1199.99	Seagate	9216

➤ *Figure 2*

The all-in-one approach means that we create a single huge table for all items, with columns for all possible attributes of any possible item we wish to store. Most likely one column will be a code defining the category of the item. In Figure 1, the `Category` column tells us what kind of piece we are looking at: `MON` for monitors, `CPU` for CPUs, `HDD` for hard drives, etc.

I also call this the 'beat it with a hammer until it fits' approach. While this strategy does give us ready access to any item in our inventory, it also requires that doing pretty much anything with that item requires special coding to examine the `Category` column and pull the applicable information from the rest of the columns. How would you write a query to give you a full report of all current items in the inventory? It is also very wasteful of storage space because just about every row will have columns that are not applicable to that item.

The independent entities approach calls for a separate table for each item category with similar information. It also means the shared information is repeated in each table. Figure 2 shows how we would lay out these tables.

This is a much more efficient storage plan, but does mean we have to examine one of several different tables to get details for any given inventory item. The concept of the 'category' code is still there; it's just externalized from the data. Our program will have to determine the category of a given item in order to decide which inventory table to use to look up its specifics. Also, querying the database for information on all items is still a cumbersome prospect.

Items

InventNum	Category	Price	Description
10001	MON	399.99	Sony SVGA
10021	HDD	299.99	Maxtor HDD
10075	CPU	145.00	Intel Pent II
10080	MON	599.95	Gateway SVGA
10211	CPU	115.00	AMD K6
10301	HDD	199.99	Maxtor HDD
10313	HDD	1199.99	Seagate HDD

Monitors

InventNum	ScreenSize
10001	15
10080	17

CPUs

InventNum	ClockSpeed
10075	200
10211	200

HardDrives

InventNum	Capacity
10021	4400
10301	3686
10313	9216

➤ *Figure 3*

The third approach, master-detail, combines the previous two techniques. We have a single table, containing all the shared columns and a category code and one row for every inventory item. We also have the independent entity tables as shown in Figure 2, but without the shared columns. The concept is illustrated in Figure 3. The `Items` table serves as a master control of all inventory items, with all the common information. The category code refers us to the associated detail table containing the specifics for that particular piece.

As with the other approaches, we still need a code value to tell us what type of inventory item we are looking at in order to know where to find the specific information for that item. It is still awkward for us to efficiently connect the data from the various tables regardless of its type.

This is where object oriented databases shine. To solve this problem, we define a base class called `Items`, which contains the shared information. Then we subclass each of the item categories into new classes: `Monitors`, `CPUs` and `HardDrives`. Because of subclassing, each of these new classes also have the shared data between them, *but no data is stored redundantly* (see Figure 4).

When we create an object, it is reflected in all levels up the hierarchy. Therefore, when we create a `Monitors` object, we appear to get a new row in `Monitors` and a new row in Items.

In reality, there is only one object instance in the database, reflected in all ancestor classes as well. Because of this we have multiple access paths into our data. When we want to examine all inventory items, we can scan the `Items` class. When we want to examine all the hard drives, we can scan the `HardDrives` class. If we make a change to the price of a hard drive in the `HardDrives` class, the change is automatically

➤ *Figure 4*

Items

InventNum	Price	Description
10001	399.99	Sony SVGA
10021	299.99	Maxtor HDD
10075	145.00	Intel Pent II
10080	599.95	Gateway SVGA
10211	115.00	AMD K6
10301	199.99	Maxtor HDD
10313	1199.99	Seagate HDD

Monitors

InventNum	Price	Description	ScreenSize
10001	399.99	Sony SVGA	15
10080	599.95	Gateway SVGA	17

CPUs

InventNum	Price	Description	ClockSpeed
10075	145.00	Intel Pent II	200
10211	115.00	AMD K6	200

HardDrives

InventNum	Price	Description	Capacity
10021	299.99	Maxtor	4400
10301	199.99	Maxtor	3686
10313	1199.99	Seagate	9216

reflected in the corresponding entry in `Items`. This is no surprise since, as we said, there is only one instance of the object in the database.

You could emulate this scheme by starting with the relational table shown in Figure 1 and defining views with restricted rows and columns for each of the subsets shown in Figure 2. But you would still have wasted storage space and a clumsy schema definition.

Even in the object database, when looking at any particular item, we need something to tell us what category it's in. The concept of a category code is still there, but now it is inherent to the class itself. When we have an object created in the `Monitors` class, we already know we are dealing with a monitor, even if we are accessing it from the `Items` class.

Methods

In Delphi, we can associate program code with a class in the form of a method. Most object databases will store executable code in a similar manner. Implementations vary widely, but in general a method can be written in SQL or a high-level language like C++ and stored in the object database. Client applications can fetch the code to execute on the workstation, or it might even execute on the server itself.

The usefulness of object methods should be apparent from our travels in OOP. But method storage is one of the least developed areas of object database products, and storage and retrieval of executable code may not be as streamlined as you would like. Too much depends on the specific ODBMS product you are using to comment much further on the topic this month. Next month, we'll be looking at a particular ODBMS and will go into more detail about its handling of methods.

Next Month...

In the next instalment we will continue our investigation of object oriented databases by taking a detailed look at Computer Associates' new Jasmine ODBMS. We'll see just how we can exploit this platform with Delphi.

Steve Troxell is a software engineer with Ultimate Software Group in the USA. You can contact him by email at Steve_Troxell@USGroup.com