The Database for .NET

Matisse: The best of both worlds

Matisse Software Inc.

Microsoft .NET is the platform of choice for implementing scalable and reliable enterprise applications from reusable components. But Visual Studio .NET developers building enterprise-class applications face a quandary.

The object paradigm is now the standard for modeling a wide variety of real-world scenarios. However, finding a .NET-compatible data repository optimized for such applications has become a stumbling block. While object database management systems (ODBMSs) provide the convenience of transparent persistence of objects, their client-centric architecture has not scaled well in enterprise environments. Relational database management systems (RDBMSs) do scale well, but map objects to two-dimensional relational tables. The increased overhead can reduce application performance to a crawl.

This article discusses the limits of using these two types of databases with C# and suggests a better alternative for .NET - a Post Object-Relational database that combines the best features of both. A Post Object-Relational databases shares with ODBMSs the ability to map data stored in back-end databases directly into an language-neutral representation. As with relational systems, a Post Object-Relational database can scale to meet the performance requirements of an enterprise-class .NET application.

ODBMSs: The Hidden Headache of Transparent Persistence

Finding a database that's both .NET-compatible and scalable enough for enterprise-class .NET applications has not been easy. Ideally, a .NET-compatible database should store .NET objects whose classes have been declared "persistent-capable" and can be manipulated seamlessly by C# or other .NET languages.

That has been the promise of ODBMSs, which made their appearance in the mid-1990s as a solution designed specifically for C++ objects and thus better suited for object development in C++. With ODBMSs, developers can define persistent classes in the same way transient classes are defined in the application.

An apparent advantage of object databases is the implementation of transparent persistence that automates the process of mapping persistent data objects into the data repository. With transparent persistence, you don't even have to alter your existing .NET classes to describe the persistent data that's permanently stored in the database (see Listing 1). That means you don't have to decide ahead of time, usually during the design phase, which objects to include and exclude from the database.

```
IObjectSpace oSpace;
// Create an instance of an interface
// to the ObjectSpace.
oSpace =
   ObjectSpaceFactory.CreateObjectSpace(dataSource);
// Create objects
Order odr =
    (Order) oSpace.CreateObject(typeof(Order),
                                         "Parrot", 2, 99.95);
Customer cstr =
    (Customer) oSpace.CreateObject (typeof(Customer),
                                           "John", "S", "Doe");
// Commit the changes
oSpace.BeginTransaction();
oSpace.Update(odr);
oSpace.Update(cstr);
oSpace.CommitTransaction();
```

Listing 1: Transparent persistence with an ODBMS

This convenience quickly becomes a burden, however, when developing scalable enterprise-class applications. In a typical application, objects are highly interconnected, and it's very important to know precisely which objects have been stored with the database and which have not. Consider an e-commerce application in which products, customers, and orders are all linked together (see Figure 1). The object model naturally captures the interrelationships of real-world applications. With transparent persistence, you may wind up loading an entire closure of objects even though you want to access only a single object. While the programmer wants to load only one customer, the closure of instances reachable from this object recursively loads a large portion of the database. Loading unneeded data in the Common Language Runtime (CLR) on client machines limits concurrency and scalability and increases network traffic.

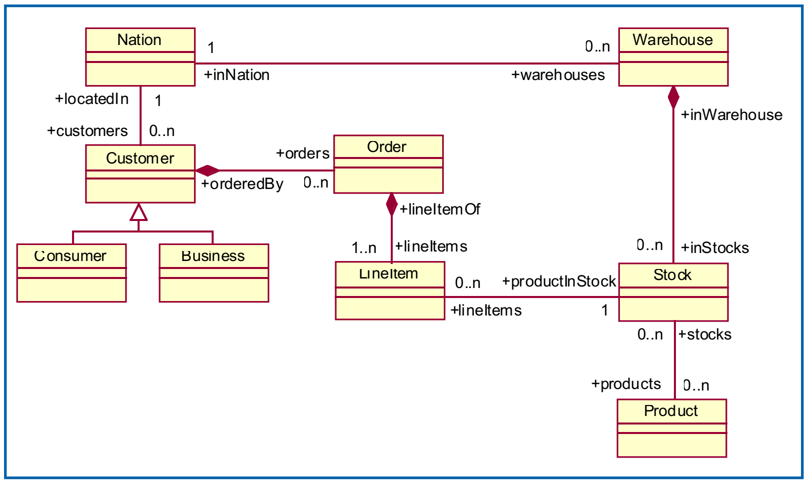

Figure 1: UML diagram of a typical e-commerce application

ODBMS's client-centric architecture promotes the implementation of the system business logic in the client application – or the middle-tier for a 3-tier architecture –, while an enterprise-class, multi-user, transaction-intensive application requires the business logic to be executed and stored into the database server to achieve top performance, enforce security and guarantee reusability. A simple customer query, for example, could move a massive amount of data – much of it unneeded – into the client application to filter objects. Such "overloading" is not a noticeable problem within a standalone environment that manipulates a small amount of data. However, in enterprise-class applications with such architecture operations rate can slow unacceptably as the system starts running under heavy computational loads, limiting the use of ODBMSs to a handful of possible applications.

While an object database is convenient to manipulate natively C# objects, the absence or limited implementation of SQL is usually unacceptable for enterprise-class applications, which require a seamless integration with database tools for reporting and business analytics capabilities.

The hard lesson, often learned at a company's expense, is that the ODBMS used to validate a pilot application must be replaced by a relational database when the system goes into production. That's the programming equivalent of a heart transplant, setting development schedules back by months. As we will see, relational databases bring their own set of problems in terms of overhead, and can require 25-50% more C# code.

RDBMSs: The Frustration of Object-Relational Mapping

.NET developers are hindered by relational databases; however, RDBMSs do have two major advantages: a long, successful track record of deployment in scalable, transaction-processing systems and a standard language, SQL. While the relational model works well enough in banking applications where the row-and-column model reflects the two-dimensional world of ledgers and spreadsheets, it has proven more limited in tracking highly interconnected information. Relationship navigation commonly used in .NET applications requires extensive use of multi-table joins. But joins are computationally intensive, and each join is computed at runtime to link information on-the-fly (see Listing 2). Reconstructing an order object with its line items from row-and-column tables requires two SQL queries and much coding. The same operation in an object database would require only one call. Moreover, relational systems require the rebuilding of relationships between objects each time they're accessed, substantially impacting performance.

In today's economy where business intelligence is key, the .NET object model provides a more powerful mechanism for capturing real-world relationships and concept commonalities. In the relational model the relationships disappear and are replaced by primary keys; foreign keys, columns, and indexes; and often by intermediate tables (see Figure 3).

In response to the demands from object developers, relational vendors have extended the relational model to support objects, much the way C++ was an object extension of C. But just as C programmers did not fully embrace C++, programmers have remained skeptical of object extensions to what is clearly not an object-oriented environment.

```
string select = "SELECT * FROM Order o
                        WHERE o.odr_id = 10608974";

IDbCommand dbcmd = conn.CreateCommand();
dbcmd.CommandText = select;
IDataReader reader = dbcmd.ExecuteReader();

// Reconstruct an instance of Order from its rows
if (reader.Read()) {
  Order odr = new Order((long) reader["odr_id"],
                      (string) reader["ship_address"],
                      (string) reader["ship_carrier"]);
}
reader.Close();
reader = null;
dbcmd.Dispose();

select = "SELECT * FROM lineItem l
           WHERE l.odr_id = 10608974 ORDER BY lt_id";

IDbCommand dbcmd = conn.CreateCommand();
dbcmd.CommandText = select;
IDataReader reader = dbcmd.ExecuteReader();

// Construct objects from rows
while (reader.Read()) {
  litem = new LineItem ((string) reader["lt_id"],
                        (long) reader["quantity"],
                        (decimal) reader["unit_price"],
                       (string)reader["delivery_mode"],
                       (string) reader["address"]);
  odr.lineItems.add(litem);
}
reader.Close();
reader = null;
dbcmd.Dispose();
```

Listing 2: The OR mapping layer adds 25% - 30% of ugly code

The underlying model of object-relational databases remains the same: rows and columns. As a result, the simplicity of the object model vanishes because classes, inheritance, and relationships must be mapped into tables - a structure ill-suited to the task. Even a simple many-to-many relationship between two classes must be expressed using intermediate tables, with two associated indexes. Therefore, a cleanly designed .NET application translated through the normalization process results in a thicket of tables that must be recombined whenever an object is called by the application. The process adds significant load, especially when executing extensive table joins.

To solve the problem of mapping objects into relational databases, a number of OR mapping tools have been created. While these tools do make it easier to develop .NET applications that use relational databases, they don't eliminate the underlying RDBMS problems of code complexity and poor performance.

Both database technologies have limitations for .NET programming. A pure object database makes sense in a standalone environment in which concurrency and network traffic are not issues. Relational databases, while accommodating transaction-processing loads, poorly simulate a true object environment.

Matisse: The Best of Both Worlds

A Post Object-Relational database like Matisse represents the best of both worlds: the ability to map objects from .NET directly to the database with the support of a standard query language (SQL-99) and the scalable, enterprise capabili-

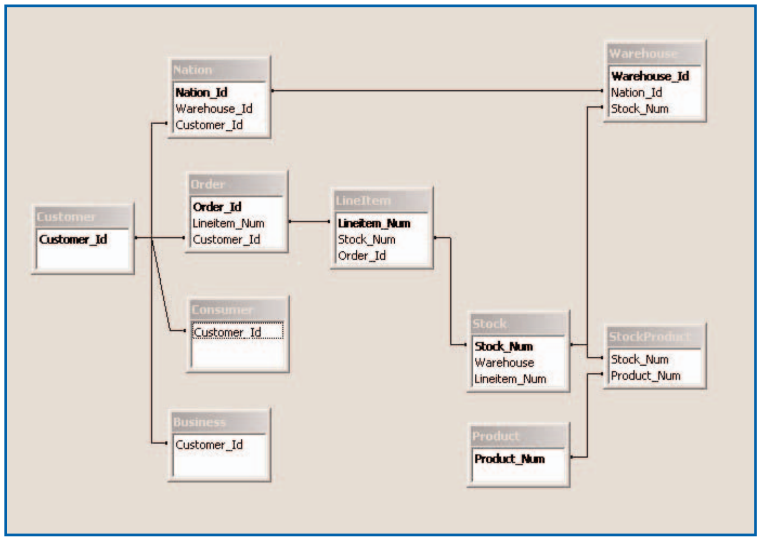

Figure 3: Relational diagram of the e-commerce application

ties implemented in relational database products. Designed from the ground up as a database server for objects, Matisse directly maps to the object model of .NET as well as other object programming languages. Because the database object model matches perfectly with .NET, you can easily define the database classes that describe real-world scenarios.

Unlike an RDBMS, Matisse preserves the original .NET data model. For example, a single class and two subclasses represent customers, consumers, and business customers, respectively. No tables are mapped back into .NET objects; no translation of any kind is needed. Unlike an ODBMS, Matisse enforces a layered design of the persistent classes. The operations to manipulate objects are explicit, enabling you to keep tight control over the data that's locked and instantiated in the CLR, seamlessly improving the application's scalability.

A Post Object-Relational database like Matisse eliminates the mismatch between the .NET and database environments, while still maintaining the scalability of server-side processing, such as relational systems. Within the .NET environment, you manipulate C# objects representing a proxy to the object in the database by means of object-to-object mapping. The proxy objects are pure .NET classes that map to those of the database schema (see Listing 3). With Matisse, the code stays compact and object-based (as in Listing 1), providing the same benefit as a first-generation ODBMS. It does not require any of the special compilation tricks or post-processing intermediate language manipulations of ODBMSs - both of which make it hard to identify the root cause of runtime errors and performance degradation.

In a typical application, classes are highly interconnected, and the graph of instances can include large portions of the database. Therefore, controlling object-locking effectively, always a challenge in enterprise-class .NET applications, is crucial to controlling the instantiation of .NET objects in the CLR. To build scalable applications, data-intensive processing needs to take place where the data sits on the server, not on the client, further reducing locking contention as well as network traffic and taking advantage of the faster processing speeds of many server architectures.

Like RDBMSs, Matisse supports the SQL-99 syntax. While SQL queries are relational in their

```
MtDatabase db = new MtDatabase(“DbName”);
db.Open();
db.StartTransaction();
Customer cstr = new Customer(db, "Dee", "O", "Haye");
Order odr = new Order(db, "Parrot", 1,  99.95);
// Link order and customer by a bi-directional
// relationship
odr.setOrderedBy(cstr) ;
db.Commit();
db.Close();
```

Listing 3: Matisse provides compactness and efficiency

syntax, they take advantage of the object paradigm by supporting inheritance, polymorphism, and true navigation. Furthermore, the query processing takes place on the server to enforce security and achieve the highest performance. Consider a broad query of two classes of customers: business and consumer. The query is issued from the client, executed on the server, with selected objects from each class retrieved to the client.

This approach gives developers full access to .NET objects through ADO.NET without having to learn a proprietary API (see Listing 4). In this listing, two customer subclasses, Consumer and Business, share properties from the parent Customer class while maintaining properties of their own. A query to locate "good customers" can combine criteria - bonus miles for home consumers, a high credit line for businesses - pulling the information simultaneously from both subclasses. Unlike an RDBMS, Matisse returns .NET objects through ADO.NET and natively supports inheritance.

While developers still benefit from the power of expression and performance of SQL queries, these queries eliminate the object-relational mapping layer to reduce source code by 25-50% and improve application performance.

Unlike first-generation ODBMSs, Matisse can be accessed through ADO.NET and ODBC drivers, both of which support the SQL-99 language, thereby taking advantage of in-house SQL expertise. Support for ODBC and ADO.NET drivers also allows IT staff to use off-the-shelf database tools without having to master SQL.

```
string query =
  "SELECT REF(c) FROM Customer c " +
  " WHERE count(orders) > 20" +
  " AND ((class Consumer).bonusMiles > 50000 OR" +
  "       (class Business).creditLine > 10000)" +
  " ORDER BY lastName";

IDbCommand dbcmd = conn.CreateCommand();
dbcmd.CommandText = query;
IDataReader reader = dbcmd.ExecuteReader();
while (reader.Read()) {
  // Instances of Consumer or Business are returned
  Customer cstr = (Customer) reader.GetObject(1)
}
reader.Close();
dbcmd.Dispose()
```

Listing 4: With Matisse, SQL queries can return objects, not only tables

Database Design for .NET: Keep It Simple

Building enterprise-class .NET applications with a Post Object-Relational database like Matisse is straightforward. Here are some considerations to make the process even smoother:

- Carefully define the object model of your persistent classes, reflecting the business model as closely as possible. That's common sense in an object environment, but is even more crucial in database applications because the way you define the model greatly impacts system performance.

- Defining the right level of granularity for your objects has a big payoff in terms of transaction throughput because only the specific queried data gets accessed and locked.

- Avoid cross-referencing persistent and transient objects as transient information can access persistent information, but not the other way around. Doing so makes the application much more complex to manage since the persistent objects loaded from the database may need to be linked to transient information that's not yet available. While a callback can also be used, it unnecessarily complicates program flow and can usually be avoided with more ordered layering of the application.

- Keep transactions as short as possible. Long transactions will unnecessarily lock data for long periods of time, making it unavailable to other business transactions.

- In some cases, data is cached by the middleware, reducing contention, but it requires "dirty reads" (reading data without locking) from the database. A way around this is to use Matisse versioning facility, which allows a consistent view of the database any time, even while users are modifying the current version.

Conclusion

A Post Object-Relational database like Matisse gives developers a new and important option when selecting a database for their .NET application. Until now, .NET developers have really had just one viable option: an RDBMS. Despite the drawbacks of the relational model, only RDBMSs solved the performance requirements intrinsic to enterprise applications.

With Matisse, Visual Studio .NET developers can demand both: a database that meets the intrinsic requirements of scalability, high transaction volumes, high-volume data transfer, and the need for high throughput, together with an object data model that more accurately represents business processes, now and in the future.

As the number of .NET applications grows, the limitations of RDBMSs and ODBMSs will become more and more apparent. Post Object-Relational databases represent the missing ingredient for broader .NET implementation, providing reliability and scalability without compromising .NET object environment.

Matisse Software Inc.
www.matisse.com
433 Airport Blvd, suite 421
Burlingame, CA 94010
650-548-2581