Realtime publishers
"Leading the Conversation"

The Essentials Series

Optimizing Database Connection Performance

sponsored by

DataDirect® TECHNOLOGIES

by Mark Scott

Copyright Statement

Managed .NET Connectivity

Connecting to a database requires a number of independent layers. The application needs to incorporate software that establishes the connection and calls to the database. A database connectivity layer needs to be in place to help manage security, communications, and data flow with the database. The database has a set of interfaces that help translate the client requests into actions within the database engine. And with the advent of .NET, the costs of managed versus non-managed code must also be considered.

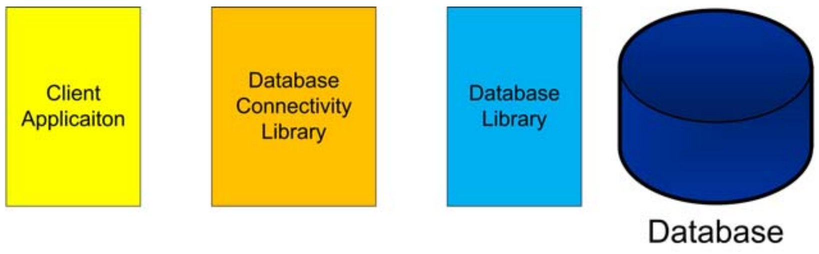

Figure 1: Database connections are layers of software.

The art of effective database connectivity is to make the distance between the database and the end use of its data as short and fast as possible. This article will explore the factors that should be considered when building an application that uses relational database technology. By considering the architecture, resource utilization, and scalability, developers can optimize their systems to provide the best performance available.

Effective Architectural Design

Microsoft's .NET architecture provides myriad advantages for developers—improved security, enhanced manageability, and simplified version control. But it also brings risks for the uninformed or the unwary. One of .NET's greatest strengths can denigrate its performance—its ability to interoperate with unmanaged code.

Like Java, .NET code operates within a virtual machine. But unlike Java's byte code, .NET uses Intermediate Language (IL) code. This allows the uncompiled code to be written in myriad languages—Visual Basic, C#, C++, Java, COBOL, APL, etc. The .NET Common Language Runtime (CLR) executes the IL. This makes the CLR source code agnostic. By compiling the source code into IL, all code execution is managed by the CLR.

However, before Microsoft developed .NET and the CLR, the programming model was the Component Object Model (COM), and its derivatives, COM+ and DCOM. To help .NET work in a COM world, Microsoft developed the Interop libraries. The Interop libraries allow .NET assemblies to interface with COM assemblies and work together. This functionality makes it simple to use .NET with existing COM applications and environments, such the Windows operating system (OS) and a great deal of legacy code.

There is, however, a cost to this convenience. The mechanisms that COM uses to create and pass events, instantiate and dispose of objects—even the format of data types—differ from .NET. In order to get the two object models to work together, the Interop libraries marshal the interfaces, events, and data passed between objects to make them compatible. This overcomes the issue of interoperation, at the expense of resource consumption and performance.

When .NET was first introduced, the quickest and easiest way to develop database connectivity software was to create an Interop layer around an existing COM library and declare the result to be a .NET connectivity library. Although this technique did work, it did not optimize the process of storing or retrieving data.

If the key to database connectivity is to minimize the time and resources required to communicate between the client application and the database, the best choice for database connectivity software will be written in 100% managed code. This will allow data to flow within the CLR without the resource consumption or additional time lag required to marshal across the Interop library.

100% managed code means that all the code used to communicate with the database operates within the CLR. A managed provider may be written to call COM components. This means that calls to these COM components need to be marshaled, costing performance, Also, when the COM component is no longer used, and it is disposed, it can fragment memory. This can lead to memory leaks.

Another key benefit of .NET code is code security. .NET was developed as Microsoft re-invented the manner in which it writes code to provide a secure platform for enterprise application development. .NET has a strong, comprehensive model for ensuring that code is secure and that identities are properly handled within the CLR. From the way variables are managed to signing assemblies, it makes it much more difficult to introduce malevolent code into the system.

For more information about Microsoft's Trustworthy Computing initiative, see http://www.microsoft.com/mscorp/twc/default.mspx.

Databases typically represent the most valuable and subsequently most highly targeted data that malefactors would want to compromise. Database connectivity software that is written entirely in managed .NET code and that leverages the many security enhancements found in the .NET platform can help to protect the security of that data from attack.

Another advantage of .NET managed code is the manner in which it handles object creation and disposal. As many who have written COM-based code have discovered, it can be difficult to ensure that COM objects are released from memory when they are no longer needed. This can lead to memory leaks and the unnecessary consumption of resources.

By writing a database connectivity suite in 100% managed code, the developer can make use of the improved facility for object life cycle management. Objects can be disposed of automatically when they are no longer in use or overtly through use of the dispose method. The automatic garbage collection provided by the .NET CLR also removes the fears of memory leaks. It automatically handles the recovery and de-fragmentation of memory allocations. If the code interoperates with existing COM objects, these benefits are lost.

Managed code libraries can improve version control. For many, the realities of "DLL hell," having the wrong version of one of several interdependent COM DLLs installed, is a painful memory. Managed .NET assemblies can be installed side-by-side. If both an Oracle and a DB2 provider needed to be installed on the same client, and one provider used a shared DLL, a different version of the DLL used in the Oracle provider could affect the DB2 provider. Data connectivity based on 100% managed .NET code avoids any use of COM DLLs. With this approach, the managed .NET assemblies that connect to Oracle would never affect the managed .NET assemblies that connect to DB2. This makes application deployment and management much simpler.

Even though providers can be installed side-by-side, it does not mean they make the best use of resources. For instance, two providers may be installed side-by-side. If those providers each call a different version of a specific COM DLL (such as a database client library), they cannot run side-by-side, since two versions of the same DLL cannot be invoked simultaneously. A better solution might be to find a provider that is 100% managed .NET code and can invoke two different versions of the same assembly at the same time.

Many developers have also been through the exercise of trying to get Visual Basic COM components to talk to C++ COM components. The data type, interface, and threading model discrepancies made this process difficult. Managed .NET code assemblies can be written in any .NET language preferred by the developer. The result is compiled to Interop Library, the result appears the same to the CLR, regardless of the language of the source code. Thus, the developers can write in the source language in which they feel most comfortable and not concern themselves with whether the result will work efficiently with the database connectivity components, or whether a compatibility interface is secretly stealing performance.

Maximizing Return from Hardware Investments

As Moore's law has proven, the power of each computer has increased exponentially. The real issue is that the cost of managing individual computers has not decreased. To leverage the power of multiple CPUs, multiple core servers and computers, each machine must do more. This allows for fewer computers to manage and helps control costs. This trend has led to the growth of virtualization. And that requires that applications, like their servers (and operations staff), do more with fewer resources.

The measure of database connectivity software then becomes how much work it can do with the resources that it is granted. This will be measured in terms of memory footprints, CPU utilization, and network bandwidth.

In terms of memory, smaller is better. There are many considerations in regard to this. The size of the database library itself is not the only concern. It is the net total of all objects that the connectivity software needs to load in order to operate. For instance, most databases provide client libraries to help developers interface with their databases. These libraries translate database interactions into the universal command set that the database uses, regardless of the connectivity method. Some database connectivity software merely provides a wrapper around this client library layer. Sometimes the layer will be written in a different level of .NET, or even in COM, requiring additional libraries to be loaded into memory. If database connectivity can avoid loading these additional components, the memory footprint can be kept smaller.

Most database connectivity software will manage pools of users, connections, and objects. The manner in which these object pools are managed will determine how well the connectivity performs, both in terms of memory consumption and CPU utilization (it takes CPU cycles to instantiate and dispose of objects).

Network bandwidth is another critical factor. Different types of workload provide different types of loads. Reporting and analytics where larger result sets are returned from a single command have a different profile than online transactional applications where high volumes of small inserts, updates, and deletes are the norm. Database connectivity software that can be tuned to manage the workload with which it is presented can make a significant difference in network bandwidth utilization and CPU cycle consumption. The fewer packet headers created (each with its own inherent overhead), the less resource constrained the server will be.

It would be nice to have a simple set of counters and a test bench that can accurately measure the impact of the database connectivity software on the system. It is fairly simple to see the memory footprint that a single assembly imposes on the system. Or to measure the network bandwidth consumed under a specific load. It is more difficult to see how these factors interact. For instance, if the available RAM is consumed and memory is swapped to the hard drive, performance drops precipitously. Suddenly, there is an increase in hard disk IO and a corresponding spike in CPU utilization. Conversely, the network bandwidth drops because the system needs time to catch up before it can handle more packets. All these factors must be considered holistically to accurately determine the effectiveness of the database connectivity software

In the end, the only test that really matters is the resource consumption under real-world workloads. The way the system operates with a realistic number of connections and simultaneous calls to the database is much more telling than the typical artificial load tests. Given a realistic load, the next challenge is determining how the resources are being consumed and released. Database connectivity software that can report on its own resource consumption can prove invaluable in this process. If the software also provides a mechanism to tune its resource utilization to the demand of the real-world workload, the resources consumed communicating to the database can be optimized and the host system can do more work with the resources it has at hand.

Scaling to Meet the Needs of the Organization

IT is always placed in a difficult position. People want more information, processed in more ways to make it easier to use, but they want to spend less and less to get it. Organization that stored gigabytes of data just 2 years ago, are purchasing terabyte SANs and asking themselves if it is enough space. As information becomes ubiquitous—shared with internal employees, clients, partners, and other systems—the connections to those databases must grow without compromising reliability or security.

Resource utilization needs to be optimized not just within a single server but across the organization. Optimization of network bandwidth quickly becomes a critical enterprise issue. As more clients demand database access and generate traffic, minimizing that traffic with each call becomes increasingly important.

Increased demand for connectivity will also increase contention for shared resources. Database connectivity components that can minimize resources on a shared virtual host (using less memory, CPU cycles, etc.) are better citizens on those servers. Effective connection pooling can help reduce load on the backend database server and conserve memory resources on that server.

Data that becomes mission critical is as limited by the database connectivity software as it is by the backend database. Software that can react appropriately to database clusters, mirrors, shared databases, and other techniques for ubiquitous database availability will ensure that data is available 24 × 7 × 365.

As the need for database connectivity grows, servers will be set in farms and the traffic will be load balanced across those servers. Database connectivity software must be engineered to handle the requirements of load balanced access to the database. Effective caching and management of the object pools can help in this.

Optimizing Database Connectivity

To optimize database connectivity, an application should store and retrieve data from the database as quickly as it can. It should optimize the use of memory, CPU cycles, and network bandwidth. And with all optimizations, there will be tradeoffs. Good software will be tunable, so the scarcest resources in the system can be preserved while still providing acceptable performance.

Database connectivity may seem like a small issue in the context of a large enterprise solution. But in a very real sense, the database is nothing to the client but the database connectivity software used to connect to it. In the .NET world of managed code, connectivity software that is 100% managed can be safer, easier to maintain, and more performant than software that uses a mixed .NET/COM model. That software should efficiently use the memory, CPU cycles, and network bandwidth. It should be tuned to the workload that it is presented, and provide facility for monitoring and troubleshooting its own performance. It should also provide scalability features to allow it to work in the load balanced, always available world of mission-critical database applications.