Realtime publishers
"Leading the Conversation"

The Essentials Series

Optimizing Database Connection Performance

sponsored by

by Mark Scott

Article 1: Managed .NET Connectivity 1
Effective Architectural Design 1
Maximizing Return from Hardware Investments 4
Scaling to Meet the Needs of the Organization 5
Optimizing Database Connectivity 6
Article 2: Open Database Connectivity 7
Designed to Perform 7
Making Efficient Use of Resources 9
Reaching Out Into the Enterprise 10
Reaching Optimal Performance 11
Article 3: Java Database Connectivity 12
Designed to Perform 12
Getting the Most from Server Resources 14
Built for Enterprise Use 15
Optimizing the Solution 16

Copyright Statement

© 2008 Realtime Publishers, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtime Publishers, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtime Publishers, Inc or its web site sponsors. In no event shall Realtime Publishers, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtime Publishers and the Realtime Publishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtime Publishers, please contact us via e-mail at info@realtimepublishers.com.

Article 1: 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.

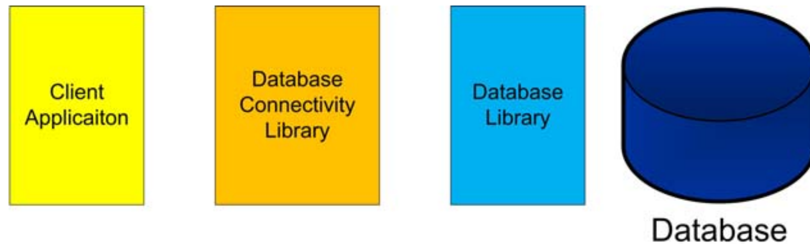


Figure 1: Database connections are layers of software.

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.

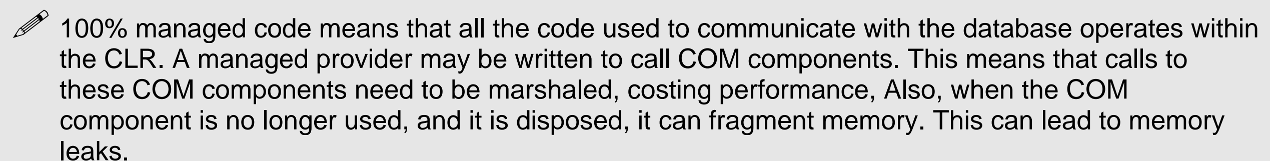
Like Java, .NET code operates within a virtual machine. But unlike Java's byte code, .NET uses Intermediate Language (IL) code. This allows the uncompiled code to be written in myriad languages—Visual Basic, C#, C++, Java, COBOL, APL, etc. The .NET Common Language Runtime (CLR) executes the IL. This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

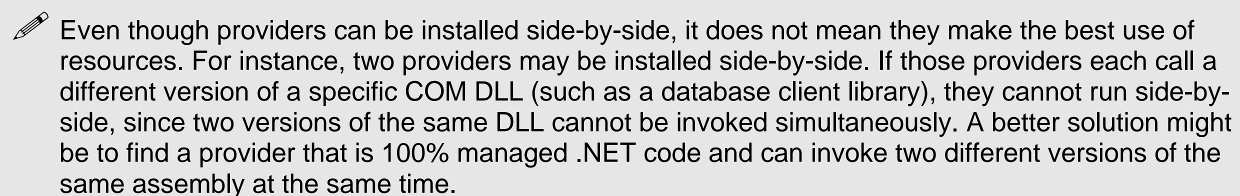
For more information about Microsoft's Trustworthy Computing initiative, see http://www.microsoft.com/mscorp/twc/default.mspx.

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.

Article 2: Open Database Connectivity

Since the SQL Access Group created the Call Level Interface, ODBC has become the most ubiquitous method for connecting to relational database sources (Source: http://support.microsoft.com/kb/110093). ODBC was developed to allow programmers to access relational data in a uniform manner, regardless of the database backend. ODBC translates those generic commands into the specific esoteric commands of the database backend, so the quality of the driver directly determines the performance of the database connectivity layer.

The performance of these drivers varies widely. By understanding the architecture of the ODBC driver, the manner in which it utilizes computer resources, and the scalability of the component, you can make intelligent decisions in choosing and implementing the best drivers for any given enterprise architecture. And with the explosive growth of data access, even small performance gains for each instance can make a big difference in the operation, cost, and maintainability of an organization's data access infrastructure.

Designed to Perform

ODBC is based on a pair of components working at the client application. An ODBC driver manager provides a uniform interface for connecting to and interacting with relational data sources. The architecture is flexible enough that different flavors of SQL can be passed to the server, yet uniform enough that the coder knows what steps he or she must take to work with the database, regardless of what that database might be.

The real magic exists in the ODBC driver. The ODBC driver takes the common commands issued by the developer and translates that to the specific commands used by the target database. This layer of abstraction allows the databases to be developed independent of a constricting standard. They can develop their own individual strengths and new features, without limiting themselves to the restrictions of the standard.

Many database vendors have taken the approach of building ODBC drivers for their databases on top of their database client libraries (see Figure 1), an approach that makes it easier to quickly build ODBC drivers. Driver developers can simply focus on translating ODBC function calls and standard SQL syntax to the client API and database SQL syntax without worrying about coding to the lower-level API of the database itself and managing the headaches of communication over the network.

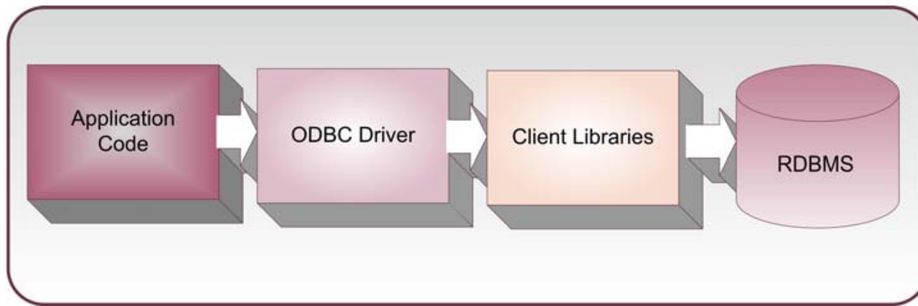


Figure 1: ODBC driver built on top of database client libraries.

An ODBC client that makes direct calls to the database without the use of the client libraries can run more efficiently. By making direct calls to the lower-level interfaces, the ODBC client can expose and leverage the full range of functionality provided by the RDBMS. This allows the data connectivity layer to exploit the full power of the underlying data store. Use of these lower-level calls opens the door for optimization of the database calls. A well-written client speaking more directly to the database will use less memory, operate more efficiently, and provide enhanced facilities for tuning the database calls to the specific workload.

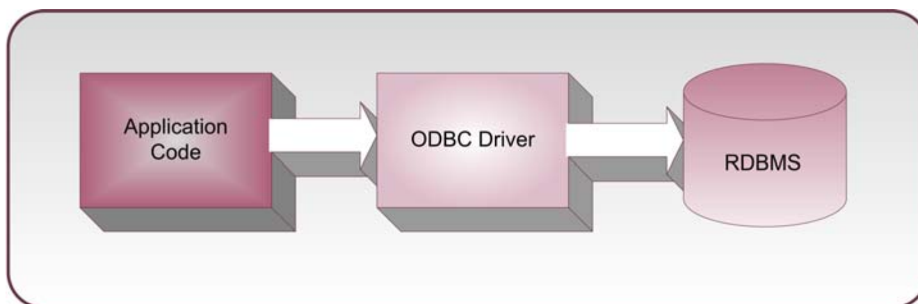


Figure 2: An ODBC client that makes direct calls to the database.

The application platform needs to be considered as well. 64-bit applications provide an extended memory model—very welcome for applications that move large amounts of data such as ETL, data warehousing, and business intelligence applications. The ODBC drivers used in these applications need to run in the same architecture as their client application: x86, x64, IA64, and so on. If the application runs in multiple chip architectures, then multiple versions of the ODBC driver must be utilized.

Some tools restrict full-featured development to a single architecture, such as x86, but ultimately allow the finished product to run in a different architecture. Doing so requires the use of another version of the ODBC driver, one that runs in the architecture in which the final application ultimately runs. Having a complete set of ODBC drivers that offer compatible features across all architectures can simplify development and reduce headaches when the application moves to deployment in production.

Making Efficient Use of Resources

In most enterprises, the initiative is to do more with fewer servers. Power consumption, rack space, and the cost of maintaining multiple servers—each with their own maintenance contract, has caused many organizations to get more from each server. The advent of virtualization technologies makes this even more significant.

Choosing the correct ODBC drivers, drivers that place less resource demands on the server, can make a marked difference in server consolidation projects. To choose the correct drivers, one must examine the real impact of the driver on the system to determine the demand that it generates. This can typically be measured in terms of memory, network, and CPU utilization.

Memory usage should be measured in terms of the entire stack—for example, how much RAM is available before and after all the components are loaded? If an ODBC driver needs to load database client libraries, this will increase the amount of memory consumed. If running a 32-bit application on a 64-bit system requires invoking a different subsystem, that subsystem impacts the OS. The view of system impact needs to be holistic, not limited to simply measuring the size of the binary that one portion of the system opens. This should not be measured by the size of the binaries as they sit on the disk, but rather what actually loads into system RAM during usage.

The impact of RAM consumption can determine how many simultaneous connections a Web server can host, how many processes an enterprise service bus SOA application can manage, and how much parallelism an ETL process can sustain. And if the servers are virtualized, inefficient memory usage can restrict the number of virtual servers that can operate simultaneously on the same physical server.

Network utilization is also important. The overhead for network packets can add up quickly. Connections across WAN links can be performance challenged. As the number of network hops increases, and packets are fragmented, the latency of the connection increases and the performance drops. Different workloads will have different needs. For instance, an ETL process performing a bulk load of the daily transactions through a single connection has different requirements than a Web site that spawns hundreds of connections that move very small data loads. Optimization of the network packets used to service those different loads can have a large impact.

An ODBC driver that can be tuned can help minimize the impact of these workload discrepancies. A driver that can be configured to optimize the lower-level commands it uses to communicate with the database can conduct more efficient, operations with the database.

The CPU is affected by disk I/O, network communications, and memory management. Because these factors interrelate so intimately, real-world workloads are the only accurate test of this effect. Drivers that do well with a small number of connections may falter if the workload is the simultaneous use of hundreds of connections. Understanding how the specific driver behaves, and carefully testing how it performs the type and volume of work that it will encounter in production, will help identify the correct driver for a given use.

It can be difficult to construct realistic tests, but there are many tools to help formulate such tests. The results can be quite revealing and save a lot of re-engineering that might occur if the application is placed into full-scale production and proves ineffective.

Reaching Out Into the Enterprise

The other consideration for the driver is its scalability. There are several issues to address, including resource contention, load balancing, availability, and pooling. For ODBC drivers to work in parallel, they must guard against resource contentions. All software shares dependencies. To use memory pools and common functions, the software components must be able to gain exclusive control of resources for brief periods of time. How well the ODBC driver can orchestrate the process of locking and releasing common resources will determine how many instances can operate concurrently and how well the components can scale on a shared platform.

To scale out, systems often use farms of servers to provide more servers to handle the load. Additionally, products such as Oracle RAC will use load-balanced database servers. Understanding the current and future requirements of the system can help make selection of the appropriate ODBC driver more certain. Research on the load-balancing scenarios that the driver will handle and understanding the options it provides can help reduce maintenance as the system expands.

Database availability is another issue. With differing technologies for implementing high availability, warm standby servers, data mirrors, clusters, grid computing systems, and so on, it is important to know that the ODBC driver can easily and quickly restore system operations when failovers occur. Often, systems that start with low-availability requirements escalate until the enterprise requires them all the time. Using ODBC drivers that complement the availability features and demands of the target database or databases can ease changes to meet evolving requirements.

Pooling is another area where an ODBC driver can improve performance. While balancing the need to minimize resource constrictions, the intelligent use of pools to share common objects, such as connections, can help minimize the total in-memory footprint of the database connectivity software and reduce demands for resources on the servers, both client and database. Effective management of object creation and disposal will also help reduce detrimental effects to the application, such as memory fragmentation or even memory leaks. These factors contribute to the durability and flexibility of how applications can be implemented and how they perform.

Reaching Optimal Performance

The bottom line of any database connectivity product is to minimize the overhead required to send data to and retrieve data from the database. ODBC represents a mature database connectivity technology. That means that it is stable and well understood. It also means that it has gone through the most evolutionary changes, and carries the most baggage from previous iterations. To ensure that the ODBC drivers used in an application are providing the optimal performance, consideration of the driver architecture, resource utilization, and scalability are required.

Ultimately, the architecture of the ODBC driver will lay the foundation for the resource utilization and scalability. ODBC drivers that communicate directly to the database without loading additional libraries have the potential of being the most efficient. They can reduce the total memory footprint and open the door for optimization. Drivers that operate in the required mode (32-bit versus 64-bit) and can operate natively in process with the clients can provide improved performance.

Resource utilization needs to be considered holistically. The overall memory footprint, network bandwidth consumption, and CPU cycle utilization combine to put a load on the server. Seeing this impact in context with real-world workloads will determine the cost of the database connectivity, expressed in terms of how large server systems need to be, or how many of them are required. Efficient systems can help server consolidation and virtualization projects accelerate.

Systems grow over time. The ability to scale out systems can take many forms. Placing additional requirements on existing servers, scaling out through farms and grids, and enhancing availability all have an impact on how well the systems meet enterprise service level agreements (SLAs). Choosing the correct ODBC driver up front and tuning it to the needs of the actual workload presented can help IT deliver on those commitments.

Article 3: Java Database Connectivity

Java Database Connectivity (JDBC) software opened the door for Java developers to write applications that were agnostic of the backend database store. Working on the foundation laid by Open Database Connectivity (ODBC), it opened new opportunities to simplify coding database applications in Java. As database connectivity has evolved, so has the JDBC standard. It becomes important for an individual who is developing a database connection strategy and methodology to understand this evolution. To develop an optimal JDBC connection strategy, it is necessary to evaluate the architecture of the JDBC software stack implemented. Then one can better evaluate the resources consumed by the software and the implications this consumption makes to the scalability of these connections. So informed, an organization can make better choice, and optimal use, of their JDBC connection software.

Designed to Perform

JDBC drivers come in four types, each with a different architectural approach (Source: http://java.sun.com/products/jdbc/driverdesc.html). Type-1 drivers, often called a JDBC-ODBC Bridge provide a Java code wrapper around an existing ODBC driver. This wrapper creates a bridge for the native JDBC calls. This setup simplifies connecting to any database that has a multitude of available ODBC drivers but does not perform well.

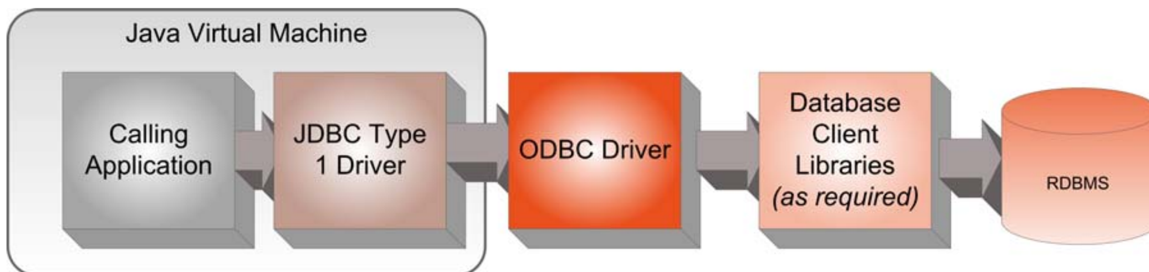


Figure 2: JDBC Type-1 driver.

Type-2 drivers, referred to as native API drivers, translate JDBC function calls into native calls of the database API. There is still overhead in translating the Java call to the native format of the database client libraries (typically written in C or C++), but less overhead than using the ODBC driver

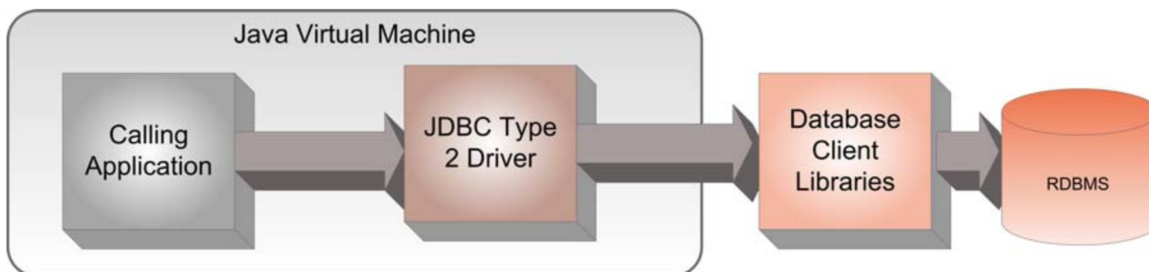


Figure 2: JDBC Type-2 driver.

Type-3 drivers, called Network Protocol drivers, were developed for application servers and middleware software services. These drivers are written entirely in Java and allow a client application to connect through a network protocol to work with the relational data source. These drivers are often part of a J2EE server and require database-specific coding. Although more efficient than calling functions written in another language, they still introduce another layer of code and translation. And as the middleware often resides on a different server or may use a different Java Virtual Machine, they introduce additional latency into each database call.

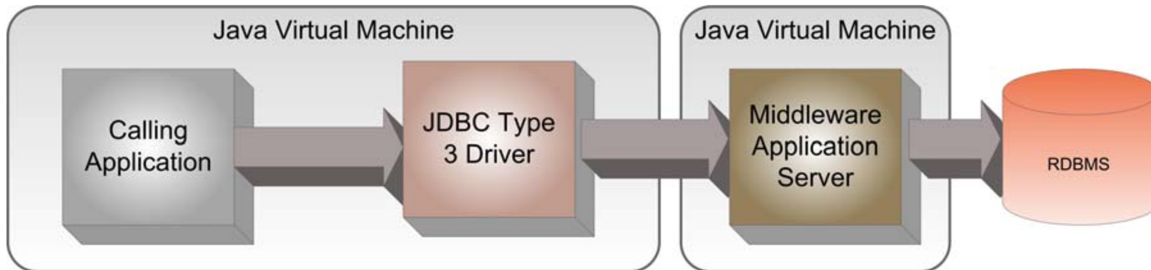


Figure 3: JDBC Type-3 driver.

Type-4 drivers, called Native Protocol drivers, are written entirely in Java to be platform independent. The driver operates within the same Java Virtual Machine (JVM) as the client, and does not incur the overhead of invoking an ODBC driver or database client library. Type 4 drivers call directly to the database API without additional layers of code or translation.

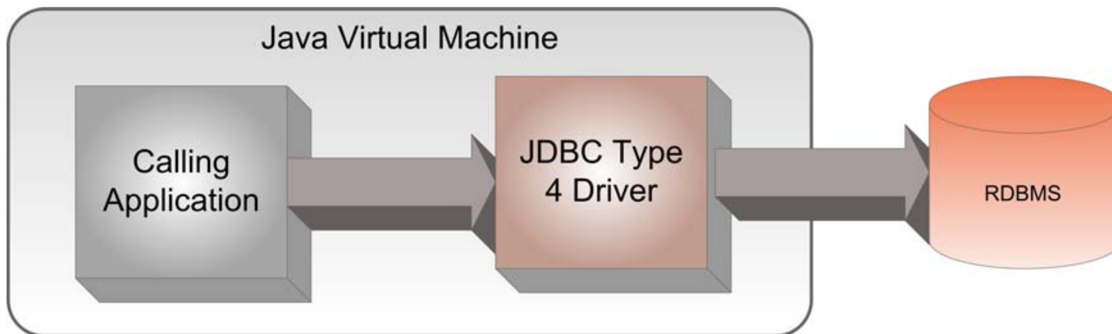


Figure 4: JDBCType-4 driver.

The key to efficiency is "minimizing" the distance from the client application to the database. Each code module that is invoked will add to the amount of memory utilized by the server. The code may require dedicated threads and additional CPU cycles to operate. When a JDBC driver must leave the virtual machine to invoke a database client library or ODBC driver, there is an even larger performance impact. The most efficient drivers will provide all the functionality required with as little code as possible.

The other issue to consider is on which JVMs the driver will operate. As the JDBC specification has evolved, JDBC drivers have been written to leverage the improving standard. Not all of these drivers have been written on or necessarily will work within the same JVM release. If the client application requires one version of the JVM and the JDBC driver requires a different version, the solution may not be viable. If it does work, there will be a great cost in resources and performance.

Another architectural consideration is whether the drivers offer the full range of functionality required by applications used in an enterprise. A line of JDBC Type-4 drivers may not offer the full range of functionality required by an organization. For instance, the Type-4 driver may not offer Kerberos authentication. When choosing drivers, it is important to ensure the driver provides the full range of functionality required by the applications it serves.

Getting the Most from Server Resources

Many organizations are trying to make IT more cost effective by keeping the server count down and getting each server to do more, pushing it nearer to its capacity. To help control server count, power utilization, and rack space, there is a strong move toward server virtualization. Minimizing the total hardware resources consumed by each individual server instance will help maximize the potential of the host server.

When virtualizing servers, the resources used by each component on the server become more critical. Choosing components that make the most efficient use of memory, CPU cycles, network bandwidth, and disk I/O will ultimately dictate the number of servers and share of workload the virtual server host can shoulder.

Memory utilization can be deceptive when using type-1 and type-2 drivers. The amount of space consumed within the JVM is not the total space the driver uses. These drivers invoke resources outside the JVM, either database client libraries or ODBC drivers, so this memory will appear within a different process. When measuring or monitoring the drivers, the total impact on the entire system must be considered.

For all types of drivers, the management of the object life cycle is important. Creating objects has a definite cost within the system and their disposal will cause memory fragmentation, so efficient use of objects and caches can make a definitive difference. This impact is not limited to memory. If ineffective creation of objects creates too much memory fragmentation and triggers more frequent garbage collection, this will impact CPU utilization as well. Efficient control of garbage utilization can minimize collection processes being triggered during times of peak utilization—times when the system can ill-afford to spend managing memory.

Network utilization becomes important, both to optimize network traffic and to minimize CPU and memory consumed in handling memory packets. Differing workloads have different needs. Servlets that perform small back-and-forth operations for an individual user have a very different profile than do Web service applications that perform data extractions for reporting or warehousing.

A driver that can be tuned to a specific workload can help optimize this traffic flow. Drivers with integrated resource monitoring can help fine tune the adjustments and to get the best use of the stack. A driver that can operate with fewer network packets will release the resources required to package and unpackage the extraneous packets.

The CPU is affected by both the network stream and memory utilization. The management of threads and object pools by the JDBC software will determine how efficiently the CPU is being utilized. Crucial to tracking this utilization is the implementation of real-world workloads. Although a driver might handle one type of workload very well, it might stumble with others. For example, a driver might perform more efficiently using a small number of connections. If the workload is the simultaneous use of hundreds of connections, and the system does not make efficient use of the object life cycle, it may not be well suited for that workload. Understanding how the specific driver behaves, and carefully testing how it performs the real work that it will encounter in production, will help identify the correct driver for a given use.

Constructing a test with a realistic number of connections and a realistic flow of data can be tedious, but doing so will reveal a wealth of information about how the drivers will really perform in production. A wealth of performance testing tools is available to help construct these tests.

Built for Enterprise Use

Applications have a way of growing in the enterprise. A small, departmental application gradually becomes an enterprise mission-critical application. A small proof of concept grows into a multi-user application that requires hundreds of simultaneous connections. And when these applications grow to the point at which they need to scale up or scale out, there is seldom time to rework them. Thinking in advance about how a particular application will scale up or scale out and planning accordingly can save a lot of rework and anguish later down the road. Scaling of components can be considered in terms of how the components scale up on a single system, how they scale out on multiple systems, and how they handle high availability.

A system originally tasked for a single, specific workload may end up servicing several. In addition, several servers may be consolidated into a virtual server. Then multiple workloads will vie for the same resources.

As already mentioned, object life cycle management is crucial here. If objects are rapidly created and then disposed, without consideration for reuse, this will tend to fragment memory and trigger more frequent garbage collection. If the components are type-1 or -2, this may also lead to memory leaks as database library components or ODBC objects are abandoned. Careful reuse of objects can help minimize these risks. Effective object pools can help in the reuse of objects. By pooling objects and caching results, both object creation and network traffic may be optimized.

Tuning the object for the workload it will handle is also important. A servlet that does simple connections with a database may appear to work quite well with a type-2 driver. The true cost of loading the database client library may not be discovered until the application is under full load. At that point, it may be difficult to reconfigure using a different driver approach. The means by which the objects are abstracted can have a serious impact. Some drivers may be limited in their use of advanced object relational model functions, such as hibernation. This can impact their reusability. It may also prevent workloads from being consolidated and require the use of more servers in the organization. This increases maintenance costs, power consumption, rack space, and a variety of other resources that extend beyond the initial cost of the driver.

Scale out is another consideration. How do the drivers operate when they work within an application server farm? How do they cooperate when operating in a more disconnected mode? The means by which the drivers can be adjusted to operate in a multi-server environment can make a real difference in the net performance gain of adding more servers to a particular task.

High availability is another consideration. With application clusters, failover clusters, database mirrors, and warm standby servers, the ability of a database driver to support the means by which data is made available to the application can have a dramatic effect. A driver may determine how long a failover will take to recover. It may limit less-expensive means of protecting data and mandate more costly routes. It may eliminate some choices altogether. Service level agreements (SLAs) may be seriously curtailed by these limitations. Consideration of the availability from beginning to end may save rework and provide economical choices as the requirements for the application increase.

Enterprise security is also a key consideration. With workloads scattered across locations, domains, and regions, using improved security mechanisms becomes more important. From a corporate compliance viewpoint, it may be mandatory. Drivers need to support the strict needs of the organization to secure its data.

Optimizing the Solution

With the number of data sources growing in most organizations, optimization of database connectivity becomes more critical. The resources used to save and retrieve data have a real dollar cost to the organization that is paid day in and day out.

By understanding the architecture of the database connectivity components used, one can better understand how to measure and predict their use of resources. By knowing which JVM and non-JVM components will load into memory, an organization can begin to see how overall resource utilization will be impacted.

By tuning components to make best use of the resources that they expend, an organization can get the best return on the computer resource expenditure. Basing this measurement on real-world workloads, one can get more from each server, each database connection, and each command.

By understanding how components will scale up on a system, an organization can project how much work a given set of server resources can do. Choosing components that scale up and scale out well can enhance an organization's ability to consolidate workloads and receive optimum return from its investment in hardware and operating servers. By knowing how the components can work with high-availability systems, an organization can commit to SLAs and meet those commitments. Careful consideration of these aspects when selecting a JDBC connectivity strategy can help an organization get the most from their applications, databases, and servers.