Realtime publishers

"Leading the Conversation"

The Essentials Series: Optimizing Database Connection Performance

Java Database Connectivity

sponsored by

DataDirect® TECHNOLOGIES

by Mark Scott

Copyright Statement

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.

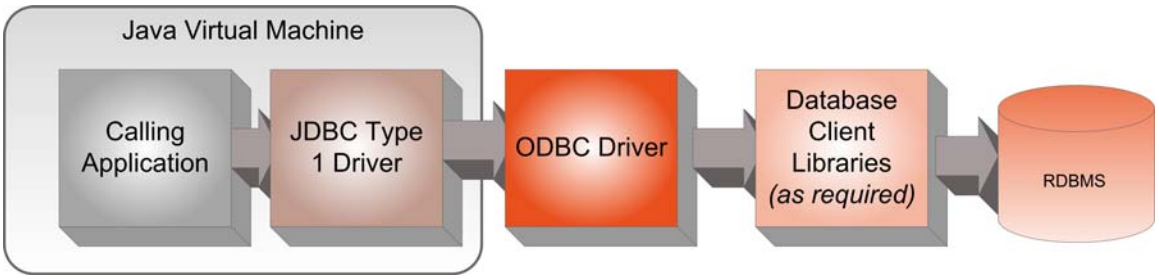

Figure 1: JDBC Type-1 driver.

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

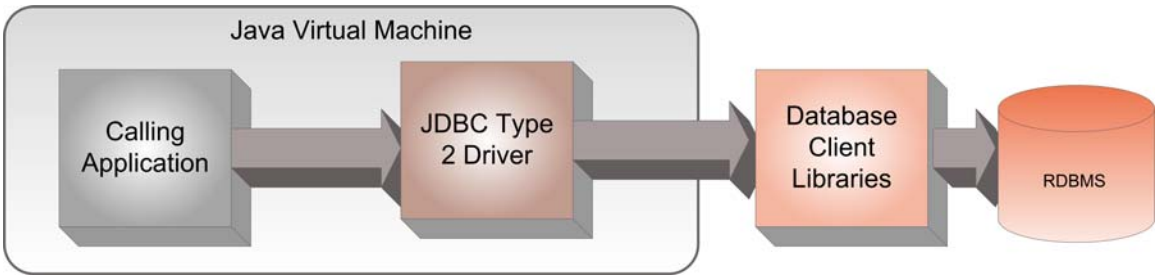

Figure 2: JDBC Type-2 driver.

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.

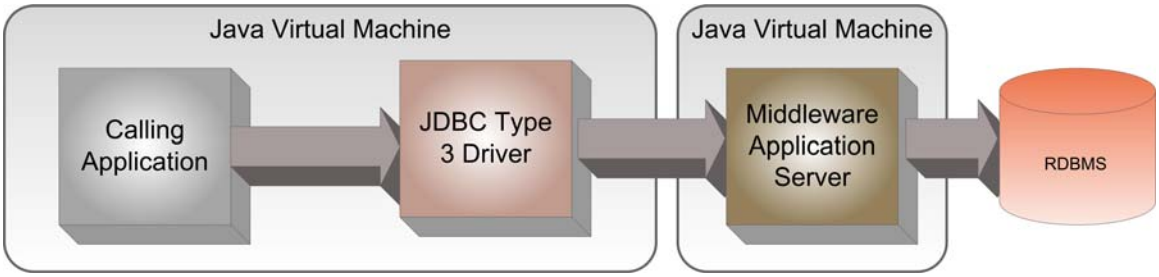

Figure 3: JDBC Type-3 driver.

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.

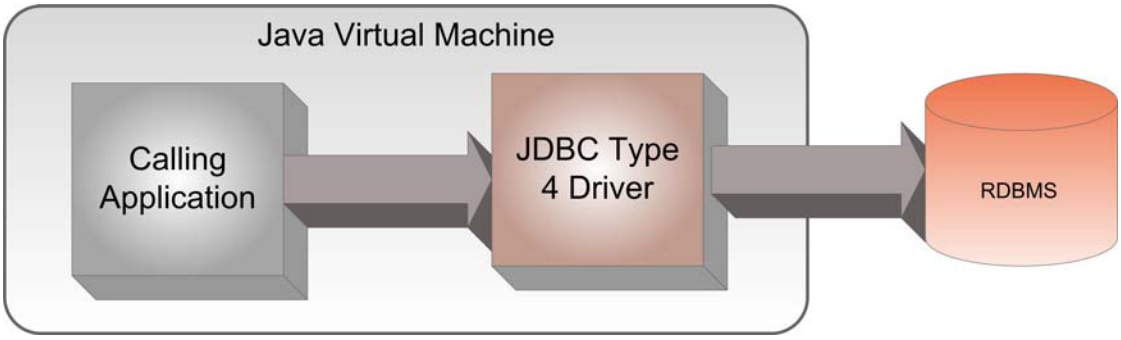

Figure 4: JDBCType-4 driver.

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.