PRENTICE HALL

THE DATA ACCESS HANDBOOK

Achieving Optimal Database Application Performance and Scalability

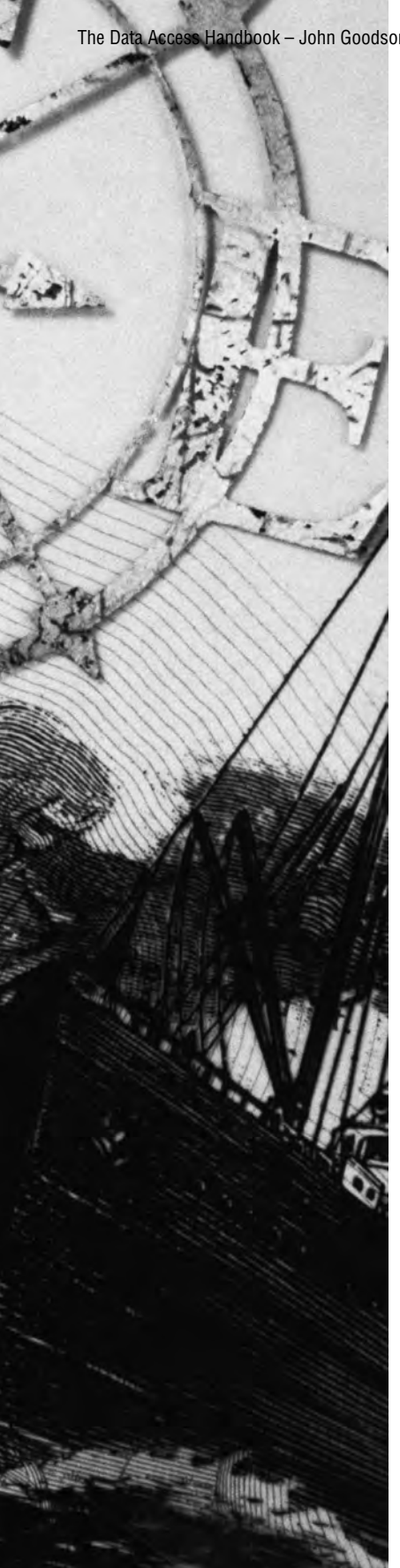
JOHN GOODSON • ROBERT A. STEWARD

The Data Access Handbook
by John Goodson and Robert A. Steward

Published March 2009

Prentice Hall

ISBN-10: 0137143931
ISBN-13: 9780137143931



CHAPTER FOUR

The Environment: Tuning for Performance

The performance of your database application, whether that is measured by response time, throughput, or scalability, is affected by many things, each of which can be a limiting factor to overall performance. In Chapter 3, "Database Middleware: Why It's Important," we explained that the database driver is only one component of your database middleware and that multiple environment layers also work with the database driver to handle the communication between a database application and the database management software. This chapter describes how those environment layers, shown in Figure 4-1, can influence performance and how to optimize performance for data requests and responses that flow through these layers. In addition, this chapter provides details about how your database driver and specific application design and coding techniques can optimize your hardware resources and relieve performance bottlenecks.

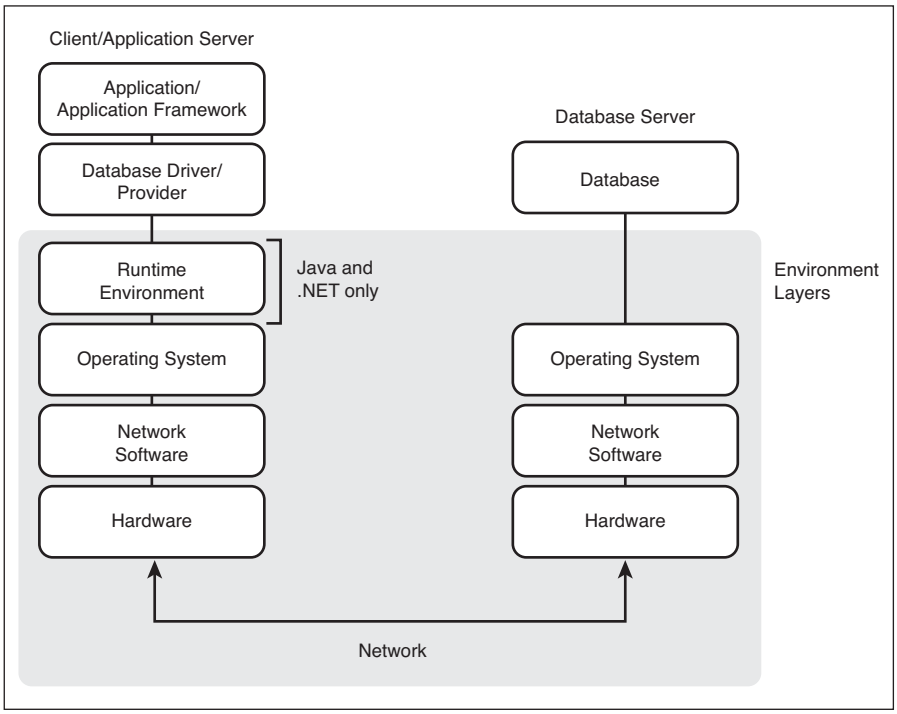


Figure 4-1 Environment layers

The influence of the environment can be significant, as shown by the following real-world example. A major business software company thoroughly tested a new database application on a local area network (LAN), and performance was acceptable according to all the benchmarks that were run. Surprisingly, when the database application was deployed in the production environment, which involved network travel over a wide area network (WAN), overall response time dropped by half. Puzzled about the performance, developers placed the actual machines used in the testing environment into the production environment; performance was still compromised. After troubleshooting the database application in the production environment, the developers discovered that the network traffic over the WAN passed through multiple network nodes with lower MTUs, which caused network packet fragmentation. See the section, "Avoiding Network Packet Fragmentation," page 98, for more information about packet fragmentation.

In this chapter, we'll talk about how the following environment layers affect performance and what you can do about it:

- Runtime environment (Java and .NET)
- Operating system
- Network
- Hardware

Runtime Environment (Java and .NET)

What do a Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR) have in common? They're both runtime environments for applications. Whereas a JVM is a runtime environment for the Java language, the .NET CLR, as part of the .NET Framework, operates as a runtime environment for multiple languages on Windows platforms. They also significantly impact the performance of your database applications.

JVM

IBM, Sun Microsystems, Oracle (BEA), and others manufacture their own JVMs. However, all JVMs are not created equal. Although vendors who produce JVMs using the "Java" trademark must adhere to the JVM specification published by Sun Microsystems, Inc., there are differences in the way those JVMs are implemented—differences that affect performance.

For example, Figure 4-2 shows the results of a benchmark that measures the throughput and scalability of a database application with different JVMs. The benchmark was run multiple times using the same JDBC driver, database server, hardware, and operating system. The only variable in this scenario is the choice of JVM. The JVMs tested were manufactured by different vendors, but were the same version of JVM and had comparable configurations. As you can see in Figure 4-2, where each line represents a benchmark run with a different JVM, the throughput and scalability of JVMs can vary significantly.

Not only does your choice of JVM matter for performance, but how that JVM is configured matters. Each JVM has tuning options that can impact your application's performance. For example, Figure 4-3 shows the results of a benchmark that used the same JDBC driver, database server, hardware, operating system, and JVM. The benchmark compares the throughput and scalability of a database application. However, the JVM was first configured to run in client mode and then configured to run in server mode. (See the section, "Client Versus Server Mode," page 82, for more information.) As you can see, the throughput

and scalability of the JVM running in server mode dramatically outperformed the JVM running in client mode.

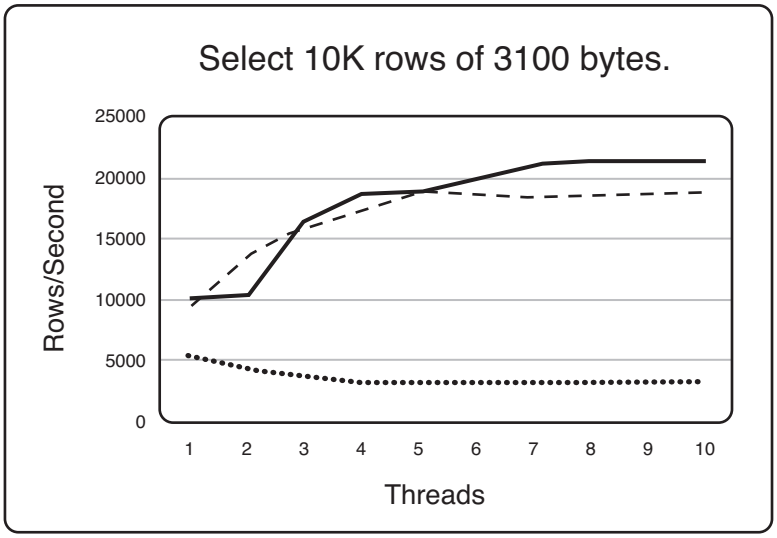


Figure 4-2 Comparing different JVMs

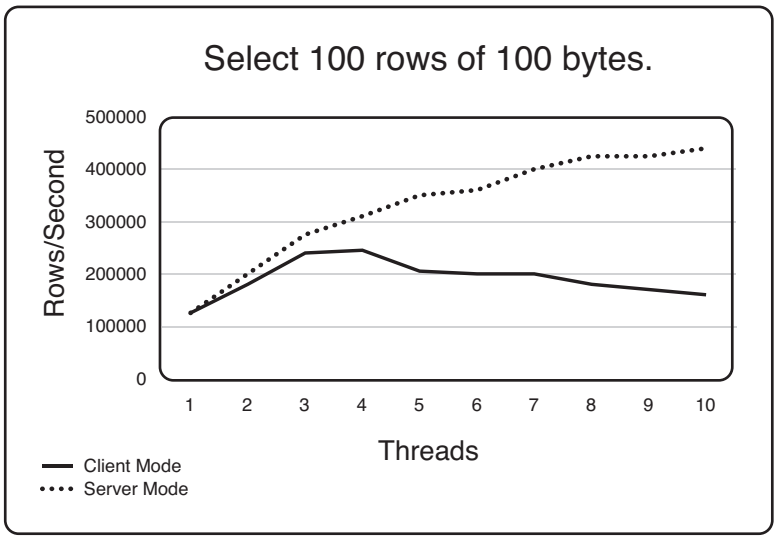


Figure 4-3 Comparing JVM configurations

You can tune the performance of your database application by setting the following common JVM options:

- Garbage collection
- Client versus server mode

Performance Tip

Choose a JVM that gives your database application the best performance. In addition, tuning options, such as those for garbage collection and client versus server mode, can improve performance.

Garbage Collection

While C++ requires direct control over when memory is allocated and freed, Java makes this process more automatic. As a Java application runs, it creates Java objects that it uses for varying lengths of time. When the Java application is finished with an object, it stops referencing it. The JVM allocates memory for Java objects from a reserved pool of memory known as the **Java heap**. This means that at any one time, the heap may have allocated memory for the following:

- Live objects that are being used by the application
- Dead objects that are no longer used (no longer referenced) by the application

Because the heap maintains memory for both types of objects and new objects are created constantly, eventually the heap runs low on memory. When this occurs, the JVM runs a routine known as a **garbage collector** to clean up dead objects and reclaim memory so that the heap has enough memory to allocate to new objects.

Why does garbage collection matter to performance? Different JVMs use different garbage collection algorithms, but most garbage collectors halt the allocation of objects while the garbage collector performs its collection routine, effectively "freezing" any application work running at the same time. Depending on the garbage collection algorithm used, this pause in work may persist as long as several seconds. When the garbage collector is finished with its collection, it lets the allocation of objects resume. For most database applications, lengthy collection pauses can negatively affect performance and scalability.

The most important options that control garbage collection are these:

- Heap size
- Generation heap size
- Garbage collection algorithm used by the JVM

The **heap size** controls how much memory is allocated to the overall Java heap. The heap size also controls how often the JVM performs garbage collection.

Finding the ideal heap size is a balancing act. When the heap size is set to a large value, garbage collection occurs less frequently, but collection pauses are longer because there's more heap to scan. In contrast, small heap sizes cause garbage collection to occur more frequently, but result in shorter pauses.

If garbage collection occurs too frequently, performance can be severely impacted. For example, suppose that your application uses a heap size that is too small to handle every live object being used by your application plus new ones that need to be created. Once the maximum heap size is reached, your application attempts to allocate a new object and fails. This failure triggers the garbage collector, which frees up memory. Your application tries again to allocate a new object. If the garbage collector failed to recover enough memory the first time, the second attempt fails, triggering the garbage collector to run again. Even if the garbage collector reclaims enough memory to satisfy the immediate request, the wait won't be long before another allocation failure occurs, triggering yet another garbage collection cycle. As a result, instead of servicing your application, the JVM constantly scavenges the heap for memory.

Performance Tip

As a general rule, try increasing the heap size so that garbage collection is not triggered as frequently, keeping in mind that you don't want to run out of physical memory (RAM). See the section, "Memory," page 107, for information about how running out of RAM affects performance. If garbage collection pauses seem unnecessarily long, try decreasing the heap size.

Older JVMs often treat the heap as one big repository, requiring the garbage collector to inspect each object in the heap to determine whether it is a dead object and can be cleaned up. Newer JVMs use **generational garbage collection** to separate objects into different memory pools within the heap based on the object's lifetime.

Some Java objects are short lived, such as local variables; others are long-lived, such as connections. Generational garbage collection divides the heap into Young and Old generations, as shown in Figure 4-4. New objects are allocated from the Young generation and, if they live long enough, eventually migrate to the Old generation. Figure 4-4 also shows another generation called the Permanent generation, which holds the JVM's class and method objects.

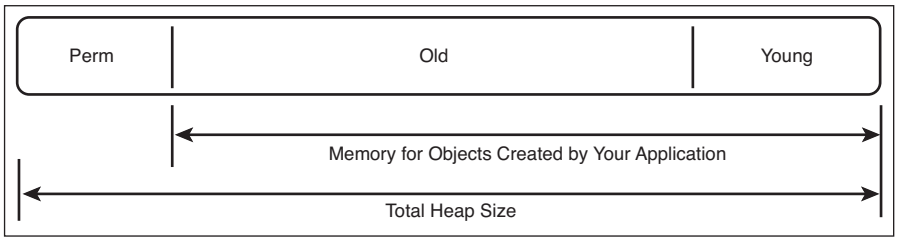


Figure 4-4 Heap generations

When the Young generation becomes full, the garbage collector looks for surviving objects while cleaning up short-lived objects. It moves surviving objects into a reserved area of the Young generation called a survivor space. If that survivor is still being used by the next collection, it's considered tenured. In this case, the collector moves the object into the Old generation. When the Old generation becomes full, the garbage collector cleans up any unused objects. Because the Young generation typically occupies a smaller heap space than the Old generation, garbage collection occurs more frequently in the Young generation, but collection pauses are shorter.

Similar to the way that the overall heap size affects garbage collection, the heap sizes of the generations affect garbage collection.

Performance Tip

As a general rule, set the size of the Young generation to be one-fourth that of the Old generation. You may want to increase the size of the Young generation if your application generates large numbers of short-lived objects.

Different JVMs use different garbage collection algorithms. A few let you tune which algorithm is used. Each algorithm has its own performance implications.

For example, an Incremental garbage collection algorithm performs its collection work a little at a time instead of trying to work its way through the entire heap, which results in shorter garbage collection pauses but reduces throughput.

Client Versus Server Mode

As a way to improve performance, many JVMs use Just-in-Time (JIT) compilers to compile and optimize code as it executes. The compiler that the JVM uses depends on the mode in which the JVM runs:

- Client mode uses a JIT compiler that is optimized for applications that are short running, need fast startup times, and require minimum memory, such as GUI applications. Many JVMs use this mode as the default.
- Server mode uses a JIT compiler that instructs the JVM to perform more extensive run-time optimization for applications that are long running and use substantial memory, such as database applications. Therefore, after startup, the JVM executes slowly until it has had enough time to optimize the code. After that, performance is considerably faster.

Performance Tip

Tune your JVM to use server mode. For database applications that run for weeks or months at a time, slower execution during the first few hours is a small price to pay for better performance later on.

.NET CLR

The CLR provides automatic garbage collection in much the same way as a JVM. When your application creates a new object, the CLR allocates memory to it from a pool of memory known as the **CLR heap**. The CLR also uses generational garbage collection. The CLR has three generations: generation 0, generation 1, and generation 2. When the garbage collector performs a collection in any of its generations, it cleans up objects that are no longer used and reclaims the memory allocated to them. Objects that survive a collection are progressively promoted to the next generation. For example, objects that survive a collection in generation 1 are moved to generation 2 during the next collection.

Unlike a JVM, the CLR doesn't provide tuning options that allow you to tune garbage collection. The CLR doesn't let you set a maximum limit on the heap size. Instead, the CLR heap size depends on how much memory can be allocated from the operating system. In addition, the CLR automatically adjusts the sizes of the generations according to its own optimization criteria.

If you can't tune garbage collection in the CLR, how can you ensure that garbage collection works in favor of your application's performance? The way your application code is designed and coded largely affects how efficiently garbage collection is performed.

Performance Tip

To optimize garbage collection in the CLR, make sure that your application closes connections as soon as the user is finished with them, and correctly and consistently use the Dispose method to free an object's resources. See the section, "Disconnecting Efficiently," page 196, for more information.

Operating System

Another factor in the environment that affects performance is the operating system. This is not to claim that one operating system is better than another—just that you need to be aware that any operating system change, no matter how minor, can increase or decrease performance, sometimes dramatically. For example, when testing an application that applied a recommended Windows update, we saw performance plummet when the database driver made `CharUpper` calls. In our benchmark, 660 queries per second throughput dropped to a mere 11 queries per second—an astounding 98% decrease.

Often, we see performance differences when running the same benchmark on different operating systems. For example, on UNIX/Linux, a database driver may use `mblen()`, a standard C library function, to determine the length in bytes of a multibyte character; on Windows, it may use the equivalent function, `IsDBCSLeadByte()`. Our benchmarks have shown that when an application used `mblen()` on Linux, the processing of `mblen()` appropriated 30% to 35% of the total CPU time. When run on Windows, `IsDBCSLeadByte()` used only 3% to 5% of the total CPU time.

It's also helpful to know which byte order, or **endianness**[1], is used by the operating system on the database client to store multibyte data in memory, such as long integers, floating point numbers, and UTF-16 characters. The endianness

[1] The term endianness was adopted from the novel *Gulliver's Travels* by Jonathan Swift, first published in 1726. In the novel, a shipwrecked castaway, Gulliver, tangled with a sovereign state of diminutive Lilliputians who were split into two intractable factions: Big-Endians who opened their soft-boiled eggs at the larger end, and Little-Endians who broke their eggs at the smaller end.

of the operating system is determined by the processor that the operating system runs on. Processors use either of the following byte-order conventions:

- **Big endian** machines store data in memory "big-end" first. The first byte is the biggest (most significant).
- **Little endian** machines store data in memory "little-end" first. The first byte is the smallest (least significant).

For example, let's consider the integer 56789652, which is 0x03628a94 in hexadecimal. On a big endian machine, the 4 bytes in memory at address 0x18000 start with the leftmost hexadecimal digit. In contrast, on a little endian machine, the 4 bytes start with the rightmost hexadecimal digit.

Big Endian

```
18000 18001 18002 18003
0x03  0x62  0x8a  0x94
```

Little Endian

```
18000 18001 18002 18003
0x94  0x8a  0x62  0x03
```

Intel's 80x86 processors and their clones are little endian. Sun Microsystem's SPARC, Motorola's 68K, and the PowerPC families are big endian. Java Virtual Machines (JVMs) are big endian as well. Some processors even have a bit in the register that allows you to select which endianness you want the processor to use.

Performance Tip

If possible, match the endianness of the operating system on the database client to that of the database server. If they match, the database driver doesn't need to perform extra work to convert the byte order of multibyte data.

For example, suppose you have an accounting application that allows you to prepare financial statements such as balance sheets, income statements, cash flows, and general ledger accounts. The application runs on Windows XP and retrieves data from a Microsoft SQL Server database running on Windows NT. The database driver doesn't need to convert the byte order of long integers

because the exchange between the machines is a match: little endian to little endian. What if you installed the application on a UNIX operating system running on a Solaris machine? You would see a drop in performance because the database driver must convert long integers retrieved from the database server from little endian to big endian, as shown in Figure 4-5. Similarly, if your application runs on a Windows machine and the database server switched to a UNIX operating system running on a Solaris machine, the database driver would need to perform byte-order conversion for long integers because of the mismatch. In many cases, you can't do anything about a mismatch, but it's helpful to know that, when all other things are equal, an endianness mismatch impacts performance.

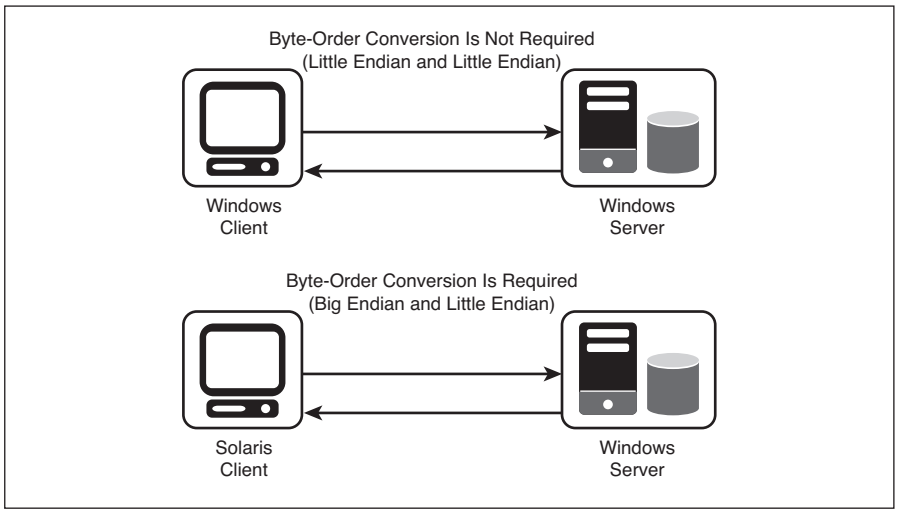


Figure 4-5 Endianness of processor determines whether byte-order conversion is required

To complicate matters, the database system doesn't always send data in the endianness of the operating system of the database server machine. Some database systems always send data either in big endian or little endian. Others send data using the same endianness of the database server machine. Still others send data using the same endianness of the database client machine. Table 4-1 lists the endianness that some common database systems use to send data.

Table 4-1 Endianness Database Systems Use to Send Data

Database Systems	Endianness
DB2	Endianness of database server machine
MySQL	Little endian
Oracle	Big endian
Microsoft SQL Server	Little endian
Sybase ASE	Endianness of database client machine

For example, suppose your application connects to an Oracle database that runs on a Windows machine. Oracle, which typically sends data big endian, must accommodate the little endian operating system it runs on and convert the byte order of multibyte data. Once again, you may not be able to change the endianness of your database client, database server, and database system to align, but it's helpful to know how endianness impacts performance if you have a choice.

Network

If your database application communicates to the database system over the network, which is part of the database middleware, you need to understand the performance implications of the network. In this section, we describe those performance implications and provide guidelines for dealing with them.

Database Protocol Packets

To request and retrieve information, database drivers and database servers transfer **database protocol packets** over a network (typically, TCP/IP).[2] Each database vendor defines a protocol for communication with the database system, a format that only that database system understands. For example, Microsoft SQL Server uses communication encoded with the Tabular Data Stream (TDS) protocol, and IBM DB2 uses communication encoded with the Distributed Relational Database Architecture (DRDA) protocol.

The way database drivers communicate to the database depends on their architecture. Some database drivers communicate to the database server directly using a database-specific protocol. Other drivers communicate using a driver-specific protocol that is translated into a database-specific protocol by a server component. Still other drivers require database vendor client libraries to com-

[2] If an application is running on the same machine as the database, the database driver uses the network in a loop-back mode or does not use the network at all and communicates directly with the database using shared memory.

municate with the database server. See the section, "Database Driver Architecture," page 55, for more information about database driver architecture.

When an application makes a standards-based API request, such as executing a `Select` statement to retrieve data, the database driver transforms that API request into zero, one, or multiple requests to the database server. The database driver[3] packages the requests into database protocol packets and sends them to the database server, as shown in Figure 4-6. The database server also uses database protocol packets to transfer the requested data to the driver.

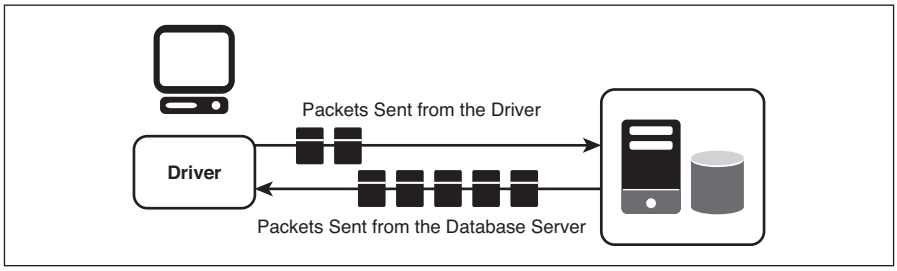


Figure 4-6 Database protocol packets

One important principle to understand: The relationship between application API requests and the number of database protocol packets sent to the database is not one to one. For example, if an ODBC application fetches result set rows one at a time using the `SQLFetch` function, not every execution of `SQLFetch` results in database protocol packets being sent to or from the database. Most drivers optimize retrieving results from the database by prefetching multiple rows at a time. If the requested result set row already exists in a driver result set cache because the driver retrieved it as an optimization on a previous `SQLFetch` execution, a network round trip to the database server would be unnecessary.

This book repeatedly demonstrates that database application performance improves when communication between the database driver and the database is optimized. With this in mind, one question you should always ask is this: How can I reduce the amount of information that is communicated between the database driver and the database? One important factor for this optimization is the size of database protocol packets.

The size of database protocol packets sent by the database driver to the database server must be equal to or less than the maximum database protocol packet size allowed by the database server. For example, if the database server accepts a

[3] Generally, we state that the database driver sends the database protocol packets to the database server. However, for drivers that have a client-based architecture, this task is performed by the database client (Net8 for Oracle, for example).

maximum packet size of 64KB, the database driver must send packets of 64KB or less. Typically, the larger the packet size, the better the performance, because fewer packets are needed to communicate between the driver and the database. Fewer packets means fewer network round trips to and from the database.

Note

Although most database applications experience better performance when sending and receiving fewer packets, this is not always the case, as explained in the section, "Configuring Packet Size," page 92.

For example, if the database driver uses a packet size of 32KB and the database server's packet size is configured for 64KB, the database server must limit its packet size to the smaller 32KB packet size used by the driver. As shown in Figure 4-7, this increases the number of packets sent over the network to retrieve the same amount of data to the client.

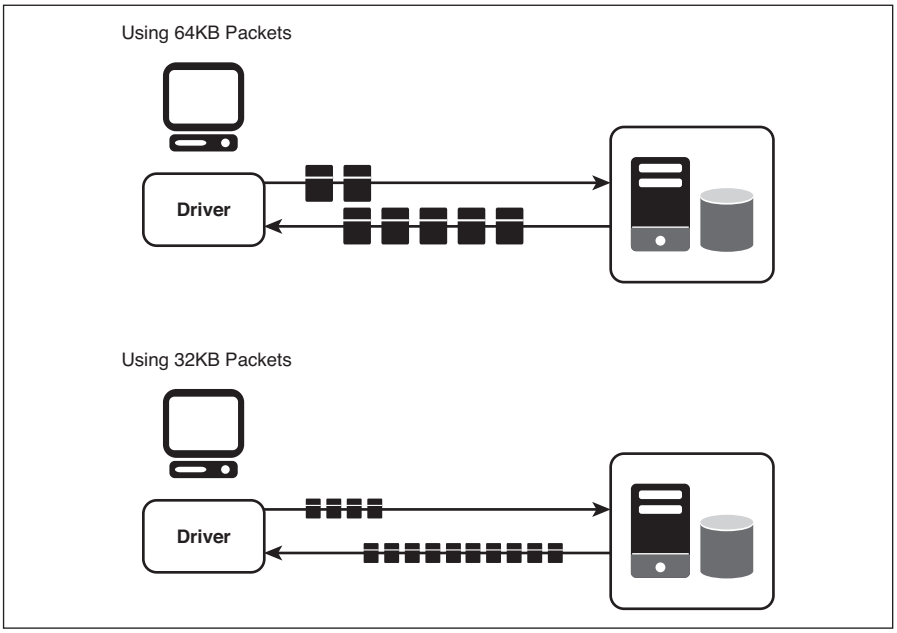


Figure 4-7 Packet size affects the number of database protocol packets required

The increase in the number of packets also means an increase in packet overhead. High packet overhead reduces throughput, or the amount of data that is transferred from sender to receiver over a period of time.

Why does packet overhead reduce throughput? Each packet stores extra bytes of information in the packet header, which limits the amount of data that can be transported in each packet. The smaller the packet size, the more packets are required to transport data. For example, a 64KB packet with a packet header of 30 bytes equals a total of three 32KB packets, each with 30-byte packet headers, as shown in Figure 4-8. The extra CPU required to disassemble packets for transport and reassemble them when they reach their destination reduces the overall transmission speed of the raw data. Fewer packets require less disassembly and reassembly, and ultimately, use less CPU.

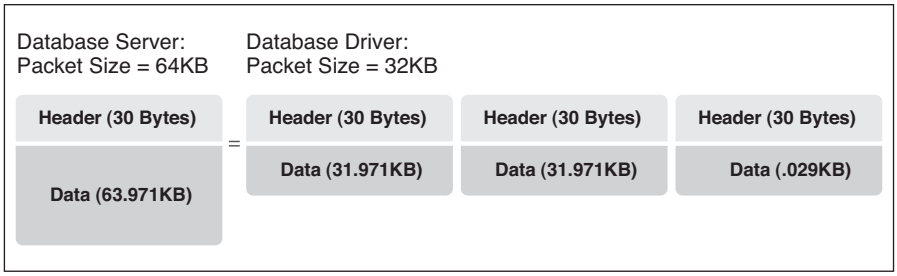


Figure 4-8 64KB database protocol packets compared to 32KB packets

Network Packets

Once database protocol packets are created, the database driver hands over the packets to TCP/IP for transfer to the database server. TCP/IP transfers the data in **network packets**. If the size of the database protocol packet is larger than the defined size of the network packet, TCP/IP breaks up the communication into even smaller network packets for transmission over the network and reassembles them at their destination.

Think of it like this: The database protocol packet is like a case of diet soda, which can be too difficult to carry over a long distance. TCP/IP breaks up that case into four 6 packs, or network packets, that can be easily carried over the network. When all four 6 packs reach their destination, they are reassembled into a case.

Similar to database protocol packets, the fewer the network packets, the better the performance. In contrast to database protocol packets, you can't configure the size of network packets.

Each network node (any machine connected to the network such as a client, server, router, and so on) has at least one network adapter for each network it connects to. The network packet size is determined by a **maximum transmission unit (MTU)** setting[4] for the network adapter in the operating system of the sending network node. The MTU is the maximum packet size that can be sent across a particular network link and is a characteristic of the network type. By default, the MTU setting is set to the MTU of the network type. You can set the MTU setting to another value, but that value cannot exceed the MTU of the network type.

For example, if the network packet size is 1500 bytes (MTU for Ethernet networks), TCP/IP breaks up the database protocol packet into as many 1500-byte network packets as needed to transfer the data across the network, as shown in Figure 4-9.

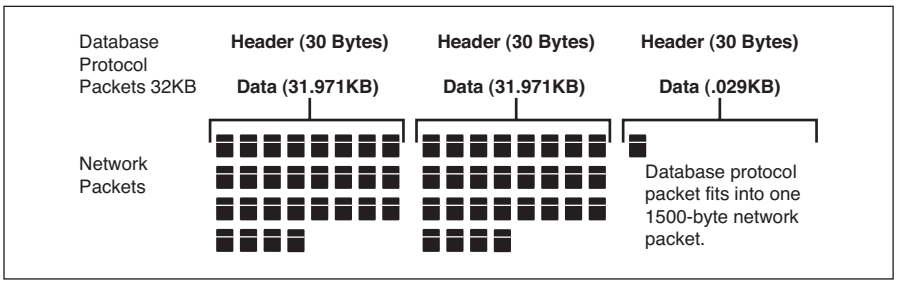


Figure 4-9 Database protocol packets divided into network packets

See the section, "Understanding Maximum Transmission Unit (MTU)," page 99, for details about how MTU affects network packets.

Database drivers and database servers only deal with database protocol packets, not network packets. Once network packets reach their destination, such as a database server, the operating system of the database server reassembles them into database protocol packets that deliver the communication to the database. To understand how this happens, let's take a closer look at network packets and how a network such as TCP/IP works.

Like a busy highway with a limited number of lanes, a network has a limited amount of bandwidth to handle network traffic between computers. By breaking up communication into network packets, TCP/IP can control the flow of traffic.

[4] The name of this setting depends on the operating system. Refer to your operating system documentation for details.

Like cars merging onto a highway, network packets can merge into traffic along with packets sent from other computers instead of hogging the road, so to speak.

The header of each network packet contains information about the following:

- Where the network packet comes from
- Where the network packet is going
- How the network packet will be reassembled with other network packets into a database protocol packet
- How to check the network packet content for errors

Because each network packet essentially contains its own shipping instructions, not all network packets associated with a single message may travel the same path. As traffic conditions change, network packets may be dynamically routed through different paths in the network. For example, if Path A is overloaded with traffic, network packets may be routed through Path B, reducing the congestion bottleneck as shown in Figure 4-10.

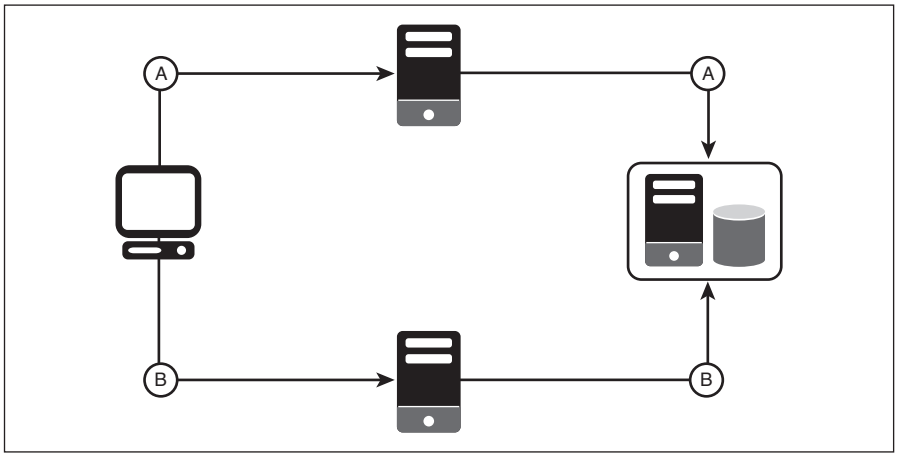


Figure 4-10 Network packets may travel different paths as a result of dynamic routing

Network packets can even arrive at their destination out of sequence. For example, network packets traveling Path B may arrive at their destination before those traveling on Path A. When all packets reach their destination, the operating system of the receiving computer reassembles the network packets into a database protocol packet.

Configuring Packet Size

Remember that larger packet sizes typically provide the best performance because fewer packets are needed to retrieve data, and fewer packets means fewer network round trips to and from the database. Therefore, it's important to use a database driver that allows you to configure the packet size of database protocol packets. See the section, "Runtime Performance Tuning Options," page 62, for more information about performance tuning options to look for in a database driver. In addition, many database servers can be configured to use packet sizes that are larger than the default.

If network packets are really the way that data is transported over the network and the MTU of the network controls the size of network packets, why does a larger database protocol packet size improve performance? Let's compare the following examples. In both examples, a database driver sends 25KB of data to the database server, but Example B uses a larger database protocol packet size than Example A. Because a larger database protocol packet size is used, the number of network round trips is reduced. More importantly, actual network traffic is reduced.

Example A: Database Protocol Packet Size = 4KB

Using a 4KB database protocol packet, as shown in Figure 4-11, the database driver creates seven 4KB database protocol packets (assuming a 30-byte packet header) to send 25KB of data to the database server (6 packets transporting 3.971KB of data and 1 packet transporting 0.199KB of data).

Figure 4-11 4KB database protocol packet size

If the MTU of the network path is 1500 bytes, as shown in Figure 4-12, the database protocol packets are divided into network packets for transport across the network (total of 19 network packets). The first 6 database protocol packets are each divided into three 1500-byte network packets. The data contained in the last database protocol packet fits within one 1500-byte network packet.

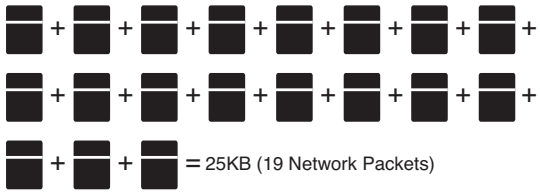


Figure 4-12 4KB database protocol packets divided into 1500-byte network packets

Now let's look at Example B, which uses a larger database protocol packet size.

Example B: Database Protocol Packet Size = 32KB

Using a 32KB database protocol packet, the database driver only needs to create a single 32KB database protocol packet to send 25KB of data to the database server (assuming a 30-byte packet header), as shown in Figure 4-13.

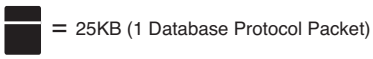


Figure 4-13 32KB database protocol packet size

If the MTU of the network path is 1500 bytes, as shown in Figure 4-14, the single database protocol packet is divided into 17 network packets for transport across the network, a reduction of 10% when compared to Example A.

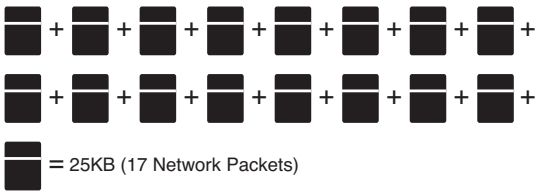


Figure 4-14 32KB database protocol packets divided into 1500-byte network packets

Although a larger packet size is typically the best choice for performance, this isn't always the case. If your application sends queries that only retrieve small result sets, a small packet size can work well. For example, an ATM banking application typically sends and receives many packets that contain a small amount of data, such as a withdrawal amount, a deposit amount, and a new balance. A result set that contains only one or two rows of data may not completely fill a larger packet. In this case, using larger packets wouldn't improve performance. In contrast, a reporting application that retrieves large result sets with thousands of rows performs best when using larger packets.

Performance Tip

If your application sends queries that retrieve large amounts of data, tune your database server packet size to the maximum size, and tune your database driver to match the maximum size used by the database server.

Analyzing the Network Path

Often, we talk about database access as if the client is always local to the database server, perhaps in the same building connected by a LAN. However, in today's distributed computing environment, the reality is that a user working from a client desktop in New York may retrieve data stored in a database that is located in California, or Europe, for that matter.

For example, a database application may send a data request that travels across a LAN, often through one or multiple routers, across a WAN, and through more routers to reach the target database. Because the world's most popular WAN is the Internet, an application may also need to communicate through one or multiple Internet service provider (ISP) routers. Then the data that is retrieved from the database must travel back along a similar path before users even see it on their desktops.

Whether your database application accesses a database server locally on a LAN or your data requests follow a more complicated path, how do you determine if network packets associated with your database application are using the most efficient path?

You can use the `tracert` command (Windows) and the `traceroute` command (UNIX/Linux) to find out which network nodes the network packets travel through on their way to a destination. In addition, by default, these commands display a sampling of the **latency**, the time delay it takes to make a network round trip to each node along the traced path.

Example A: Using the `tracert` Command on Windows

This example traces the path that network packets take from a database client in North America to a database server in Europe. Let's execute the `tracert` command:

```
tracert belgserver-01
```

Notice that the trace report shows that network packets make three network hops. (The fourth network node in the list is the destination.)

```
Tracing route to belgserver-01 (10.145.11.263)
over a maximum of 30 hops:

  1     <1 ms     <1 ms     <1 ms     10.40.11.215
  2      1 ms      3 ms      3 ms     10.40.11.291
  3    113 ms    113 ms    113 ms     10.98.15.222
  4    120 ms    117 ms    119 ms     10.145.16.263
```

Example B: Using the `traceroute` Command on UNIX/Linux

This example traces the path that network packets take on the return trip. Let's execute the `traceroute` command:[5]

```
traceroute nc-sking
```

Similar to the trace report shown in Example A, this trace report shows that network packets make three network hops.

```
Traceroute to nc-sking (10.40.4.263), 30 hops max,
40 byte packets

  1    10.139.11.215      <1 ms     <1 ms     <1 ms
  2    10.139.11.291       2 ms      1 ms      1 ms
  3    10.40.11.254      182 ms    190 ms    194 ms
  4    10.40.4.263       119 ms    112 ms    120 ms
```

[5] The `traceroute` command supports different options depending on your operating system. Refer to the command reference of your operating system documentation for command options.

After you have traced the paths going to and from the database server, let's look at what the trace report can tell you.

- Is the path taken by network packets from the client to the database server comparable to that taken on the return trip? The physical path through the network may be different in each direction, but is one path significantly slower than the other? For example, if a particular router is a bottleneck because of network congestion, you may want to change your network topology so that network packets can take a different path.
- On either path, how many network hops separate the client and database server? Can any of these network hops be eliminated? For example, if the client is assigned to a different network subnet than the database server, can the machines be reassigned to the same subnet? See the following section for details about reducing network hops.
- On either path, does packet fragmentation occur? See "Avoiding Network Packet Fragmentation," page 98, for details about detecting packet fragmentation and strategies for avoiding it.

Reducing Network Hops and Contention

There's a saying that goes something like this: "The road to success is not straight." However, when referring to data access, this adage does not necessarily apply. Shorter network paths with fewer network hops typically provide better performance than longer paths with many network hops because each intermediate network node must process each network packet that passes through that node on its way to its destination.

This processing involves checking the packet header for destination information and looking up the destination in its routing tables to determine the best path to take. In addition, each intermediate network node checks the size of the packet to determine whether the packet needs to be fragmented. On longer paths, for example, from LAN to WAN, a data request is more likely to encounter varying MTU sizes that cause packet fragmentation (see "Avoiding Network Packet Fragmentation," page 98).

A database application typically shares the network with other types of network traffic. At any one time, different users may request files and Internet content, send e-mail, use streaming video/voice, perform backups, and so on. When the traffic load is light, the network operates at top form and performance may be great. However, when large numbers of users request connections and make

other network requests at the same time, the network can become overloaded with too many network packets. If network packets sent by your database application pass through an intermediate network node that is overloaded with network traffic, application performance can be negatively affected.

Sometimes network congestion from normal business traffic is made worse by poorly planned network topology or bandwidth changes. For example, if network packets are forced to pass through a single gateway router to reach their destination, packets must wait in the router's queue for processing, causing a packet backup at the gateway. In this case, is it possible to change your network topology by adding additional router access to the destination network? Similarly, differences in bandwidth from LAN to WAN can cause a communication slowdown, much like a 4-lane highway merging into a 2-lane highway.

One way to reduce network hops and network contention is to create a dedicated path for your database application using a **private data network**, which can be implemented using a network switch to a dedicated network adapter, a leased T1 connection, or some other type of dedicated connection. For example, as shown in Figure 4-15, clients have full public access to the corporate network, including e-mail and the Internet, while enjoying private direct access to the database server.

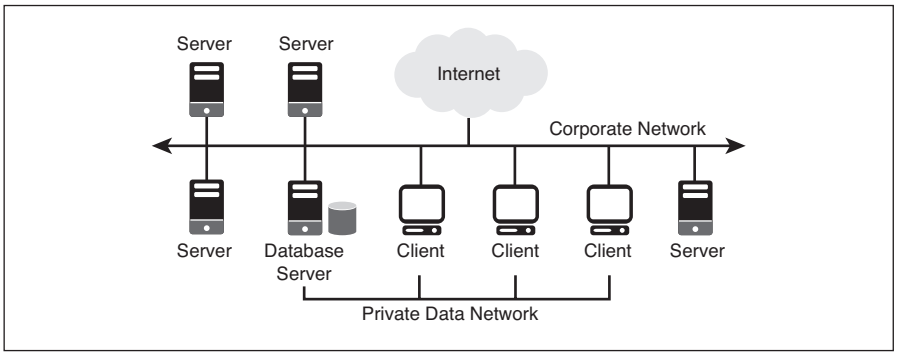


Figure 4-15 Private data network

Even when the client and database server are in proximity to one another, don't assume that network packets take a direct point-to-point path. For example, consider the case of a real-world company whose business depended on critical bulk updates that executed periodically during the course of the day. Performance was poor despite the fact that the client and database server machines were installed side by side in the same room.

The network path analysis revealed that when the application requested data, network packets associated with requests typically made as many as 17 network hops before reaching the database server. Although the client and database server machines resided in the same location, they were assigned to different corporate network subnets. In this case, reassigning the machines to the same network subnet reduced the number of network hops from 17 to 1, and the average response time for bulk updates decreased from 30 seconds to 5 seconds, an amazing performance gain of 500%.

Note

A **virtual private network (VPN)** emulates a private data network for applications that transfer data over the Internet. It doesn't eliminate network hops but provides a secure extension of a private network and reduces network contention.

Avoiding Network Packet Fragmentation

Before we go forward, let's recap some of what we've already learned about how database drivers and database servers use the network to request and send data:

- Database drivers and database servers communicate by sending database protocol packets.
- If the size of the database protocol packet is larger than the defined size of the network packet, TCP/IP divides the database protocol packets into as many network packets as needed for transmission over the network.
- The MTU is the maximum network packet size that can be sent across a particular network link and is a characteristic of the network type.
- Packet size is important for both types of packets because the fewer the packets, the better the performance.

Packet fragmentation occurs when a network packet is too large to traverse a network link as determined by the network link's MTU. For example, if a network link's MTU is 1500 bytes, it cannot transport a 1700-byte packet. An oversized packet must be divided into smaller packets that are able to traverse the link, or the communication must be re-sent using smaller packets.

In most modern systems, packet fragmentation is not automatic but occurs as a result of a process known as **path MTU discovery**, a technique for determining the **path MTU**, which is the lowest MTU of any network node along a particular network route. Because packet fragmentation requires additional communication between network nodes to negotiate the correct packet size and significant CPU processing to divide communication into smaller packets and reassemble them, it degrades performance. The following sections explain why packet fragmentation has a negative impact on performance and provide guidelines for detecting and resolving packet fragmentation.

Understanding Maximum Transmission Unit (MTU)

MTU is the maximum packet size that can be sent across a particular network link as determined by the network type. See Table 4-2 for the MTU values of some common network types.

Table 4-2 MTU Values of Common Network Types

Network	MTU
16 MB/second Token Ring	17914
4 MB/second Token Ring	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
PPPoE (WAN miniport)	1480
X.25	576

Each network node has one or multiple network adapters installed, one for each network it connects to. The operating system on each node provides an MTU setting for each network adapter. The MTU setting determines the size of network packets sent from that node. By default, this MTU setting is set to the MTU of the network type and can be set to another value, but that value cannot exceed the MTU of the network type. For example, if a network node is connected to an Ethernet network, the MTU setting for that machine's network adapter must be set to a value of 1500 (MTU for Ethernet networks) or less.

How does MTU affect network packets? Let's consider a simple example where only two network nodes, a client and database server, send and receive network packets as shown Figure 4-16. In this case, Node A has an MTU setting of 1500, meaning that it sends 1500-byte packets across the network to Node B.

Similarly, Node B has an MTU setting of 1500 and sends 1500-byte packets on the return trip to Node A.

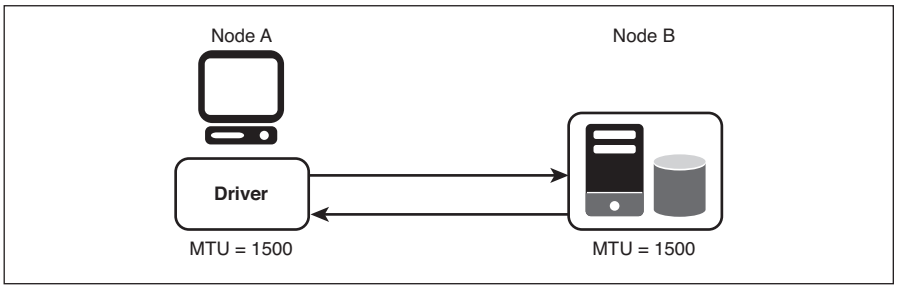


Figure 4-16 Simple example of MTU

Now let's look at a more complex example where network packets are routed by an intermediate network node to the database server, as shown in Figure 4-17. In this case, Node A has an MTU setting of 1500, Node B has an MTU setting of 1492, and Node C has an MTU setting of 1500.

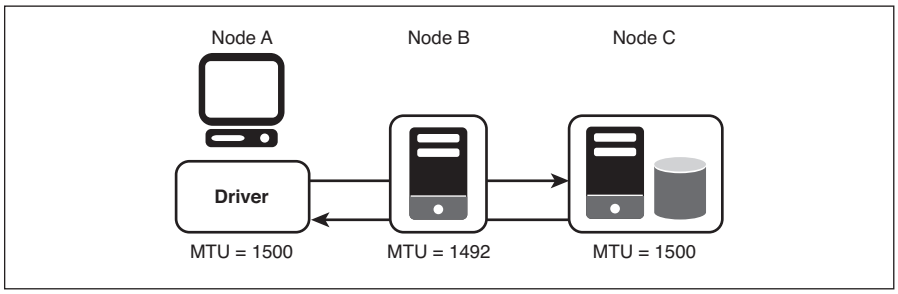


Figure 4-17 Complex example of MTU

The maximum packet size that can be sent across the network depends on the network link, or the part of the network, that the packet is being sent across, as shown in Table 4-3.

Table 4-3 Maximum Packet Size

Network Link	Maximum Packet Size
Node A to Node B	1500 bytes
Node B to Node C	1492 bytes
Node C to Node B	1500 bytes
Node B to Node A	1492 bytes

If a network node receives an oversized network packet, the network node discards that packet and sends a message to the sending network node with information about a packet size that will fit. The sending network node resends the original communication, dividing it into smaller packets. The communication required to notify the sending network node that fragmentation must occur and the resending of the communication in smaller packets increases traffic along that network route. In addition, significant CPU processing is required to divide the communication into smaller packets for transport and reassemble them when they reach their destination.

To understand how this process works, let's step through the example shown in Figure 4-18.

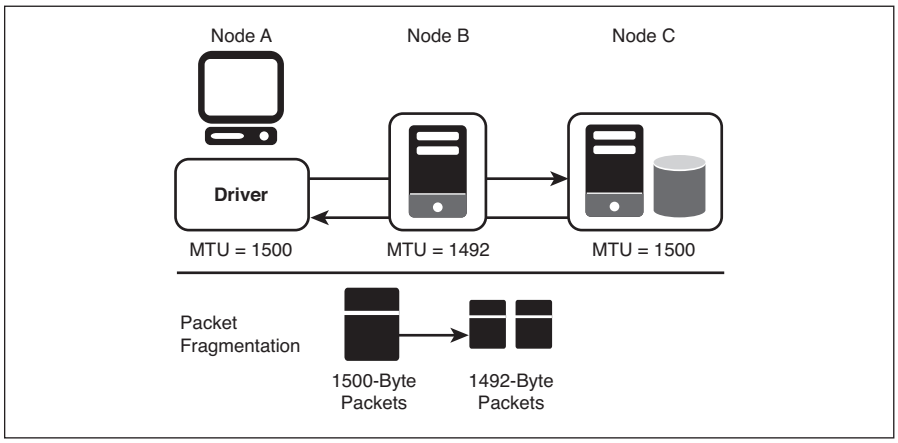


Figure 4-18 Packet fragmentation example

1. As the result of a data request, Node A sends multiple 1500-byte packets to Node C.
2. Each time Node B receives a 1500-byte packet, it discards the packet and sends a message to Node A, telling Node A that it cannot pass along a packet larger than 1492 bytes.
3. Node A resends each communication, breaking it into as many 1492-byte packets as needed.
4. When Node B receives each 1492-byte packet, it passes the packets to Node C.

Performance Tip

In most cases, you can avoid packet fragmentation by configuring the MTU setting of the client and the database server to be the same as the path MTU, the lowest MTU of any network node along the path. For example, using the scenario in Figure 4-18, if you configure the MTU setting of the client and database server to be a value of 1492, packet fragmentation would not occur.

VPNs Magnify Packet Fragmentation

Configuring the MTU setting to the path MTU doesn't always avoid packet fragmentation. For example, when VPN tunneling is used, the problem of packet fragmentation is magnified because of additional packet overhead.

VPNs are routinely used to connect remote machines over the Internet to corporate LANs, creating a secure path between two endpoints. Communication within the VPN path is encrypted so that other users of the Internet cannot intercept and inspect or modify communications. The security protocol that performs the encryption, typically Internet Protocol Security Protocol (IPSec), encapsulates, or wraps, each network packet in a new, larger packet while adding its own IPSec headers to the new packet. Often, the larger packet size caused by this encapsulation results in packet fragmentation.

For example, suppose the MTU of a VPN network link is 1500 bytes and the MTU setting of the VPN client is set to the path MTU, a value of 1500. Although this configuration is ideal for LAN access, it presents a problem for VPN users. IPSec cannot encapsulate a 1500-byte packet because the packet is already as large as the VPN network link will accept. In this case, the original communication is re-sent using smaller packets that IPSec can encapsulate. Changing the MTU setting on the client to a value of 1420 or less gives adequate leeway for IPSec encapsulation and avoids packet fragmentation.

Performance Tip

A one-size-fits-all MTU doesn't exist. If most of your users are VPN users, change the MTU setting along the network path to accommodate your VPN users. However, remember that reducing the MTU for your LAN users will cause their application performance to suffer.

LAN versus WAN

Because communication across a WAN typically requires more network hops than communication across a LAN, your application is more likely to encounter varying MTU sizes, resulting in packet fragmentation. In addition, if data has to travel over VPN within a WAN, packet fragmentation further reduces the MTU size. If you are unable to avoid packet fragmentation by setting the client and the database server to the path MTU (see "Understanding Maximum Transmission Unit (MTU)," page 99), it becomes even more important to reduce the number of network round trips between the client and server to preserve performance.

Detecting and Resolving Network Packet Fragmentation

If you don't have privy knowledge of the MTUs of every network node along the network path, how can you tell if packet fragmentation occurs? Operating system commands, such as the `ping` command (Windows) and the `traceroute` command (UNIX/Linux), can help you determine if packets are being fragmented along a particular network path. In addition, with a little persistence and detective work, you can determine the optimal packet size for the network path, a size that doesn't require packet fragmentation.

For example, suppose your client is a Windows XP machine, and data requests are made from this machine to a UNIX database server located in London. You know from the following trace report that three network hops are involved to reach the server:

```
Tracing route to UK-server-03 [10.131.15.289]
over a maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  10.30.4.241
  2    <1 ms    <1 ms    <1 ms  10.30.4.245
  3   112 ms   111 ms   111 ms  10.168.73.37
  4   113 ms   112 ms   116 ms  10.131.15.289
```

Therefore, the network path looks similar to the configuration shown in Figure 4-19. If the MTU of the client is set to a value of 1500, the client sends 1500-byte packets across the network. The MTU of the other network nodes is unknown.

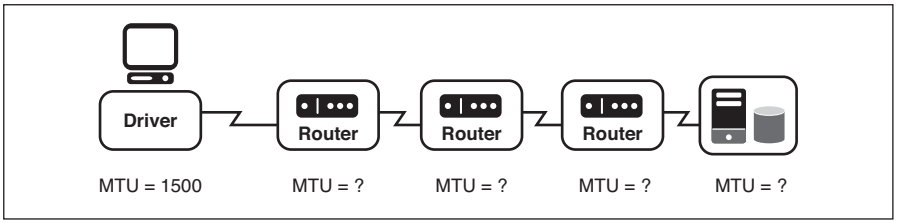


Figure 4-19 Network analysis of MTU

In the following examples, we use the `ping` (Windows) and `traceroute` (UNIX/Linux) commands to determine if packet fragmentation occurs along this network path for a 1500-byte packet, and we find the optimal packet size for the network path.

Example A: Detecting Packet Fragmentation on Windows

1. At a command prompt, enter the `ping` command to test the connection between the client and database server. The `-f` flag turns on a "do not fragment" (DF) field in the header of the packet, forcing the ping command to fail if the packet needs to be fragmented at any network node along the path. The `-l` flag sets the packet size. For example:

   ```
   ping UK-server-03 -f -l 1500
   ```

   If packet fragmentation is needed, the `ping` command fails with the following message, which indicates that the packet was not fragmented because the DF field was set:

   ```
   Packet needs to be fragmented but DF set
   ```

2. Reissue the `ping` command repeatedly, each time lowering the size of the packet in logical increments (for example, 1500, 1475, 1450, 1425, 1400, and so on) until a message is returned indicating that the command was successful.

For example, the following code shows that the `ping` command was successful when executed with a packet size of 1370 bytes:

```
Pinging UK-server-03 [10.131.15.289] with 1370 bytes of
  data

Reply from 10.131.15.289: bytes=1370 time=128ms TTL=1
Reply from 10.131.15.289: bytes=1370 time=128ms TTL=1
Reply from 10.131.15.289: bytes=1370 time=128ms TTL=1
Reply from 10.131.15.289: bytes=1370 time=128ms TTL=1

Ping statistics for 10.131.15.289:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss)
Approximate round trip times in milli-seconds:
    Minimum = 128ms, Maximum = 128ms, Average = 128ms
```

3. Once you have a packet size that works for the entire path, configure the MTU setting of the client and database server to that value (if possible).

Example B: Detecting Packet Fragmentation on UNIX/Linux

1. At a prompt, enter the `traceroute` command.[6] The `-F` flag forces the command to fail if packet fragmentation occurs. The integer sets the packet size.

   ```
   traceroute UK-server-03 -F 1500
   ```

   If packet fragmentation occurs, the command fails with the following message:

   ```
   !F
   ```

2. Reissue the `traceroute` command repeatedly, each time lowering the size of the packet in logical increments (for example, 1500, 1475, 1450, 1425, 1400, and so on) until a message is returned indicating that the `traceroute` command was successful.

[6] The `traceroute` command supports different options depending on your operating system. Refer to the command reference of your operating system documentation for command options.

The following example shows that the `traceroute` command was successful when executed with a packet size of 1370 bytes:

```
Traceroute to UK-server-03 (10.131.15.289), 4 hops max,
1370 byte packets

  1    10.139.11.215      <1 ms     <1 ms     <1 ms
  2    10.139.11.291       2 ms      1 ms      1 ms
  3    10.40.11.254      182 ms    190 ms    194 ms
  4    10.40.4.263       119 ms    112 ms    120 ms
```

Increasing Network Bandwidth

Bandwidth is the capacity of a network connection to transport network packets. The greater the capacity, the more likely that good performance will result, although overall performance also depends on factors such as latency. Increasing bandwidth is similar to widening a congested 2-lane highway into a 4- or 6-lane highway. The highway can handle more traffic, relieving bottlenecks.

Upgrading to a large-capacity network adapter is one of the easiest and cheapest investments you can make to improve the performance of your network. While bandwidth capacity has dramatically increased over the years, the costs associated with the hardware that provide it have dramatically fallen. Today, you can easily purchase a 1GB network adapter for less than $40. Assuming there are no other network constraints, upgrading from a 100Mbps network adapter to a 1GB network adapter can result in as much as a 7% to 10% performance gain. For the price and ease of effort, that's a great return on investment.

Hardware

Clearly, how your database is configured can conserve or consume hardware resources, but our focus in this section is on how database driver and specific application design and coding techniques can optimize your hardware resources and relieve performance bottlenecks in the following hardware resources:

- Memory
- Disk
- CPU (processor)
- Network adapter

In addition, we'll talk about how a hot new trend in database computing known as virtualization can magnify hardware-related performance problems.

Memory

A computer has a finite amount of **Random Access Memory (RAM)** or physical memory, and, as a general rule, more RAM is better. As the computer runs its processes, it stores code and data for quick access in blocks of RAM known as **pages**. The amount of data that can be stored on a single page depends on the processor platform.

When a computer runs out of RAM, it takes advantage of virtual memory to ensure that work processes smoothly. **Virtual memory** allows the operating system to free up space in RAM by copying pages stored in RAM that have not been used recently into a file that resides on disk. This file is called a **page file** (**swap file**), and the process of writing to the file is known as **paging**. If an application needs that page again for any reason, the operating system swaps it back out of the page file into RAM.

When RAM is being used to capacity, paging occurs more frequently. Because disk I/O is much slower than RAM, excessive paging causes a drastic performance slowdown. Excessive paging also can interfere with other processes that require the same disk, causing disk contention (see the section, "Disk," page 110, for more information). In fact, memory bottlenecks often masquerade as disk issues. If you suspect that the disk is being read from or written to excessively, the first thing you should do is rule out a memory bottleneck.

A **memory leak** can also result in excessive paging, steadily using up RAM, and then virtual memory, until the page file size reaches its maximum. Depending on how critical a memory leak is, virtual memory can be used up within a period of weeks, days, or hours. Memory leaks often are created when applications use resources, but they don't release the resources when they are no longer required.

Table 4-4 lists some common causes for memory bottlenecks and their recommended solutions.

Table 4-4 Causes and Solutions of Memory Bottlenecks

Cause	Solution
Insufficient physical memory (RAM)	Add more RAM.
Poorly optimized application code or database driver causing excessive memory use	Analyze and tune your application or database driver to minimize memory use. See "Tuning Your Application and Database Driver to Minimize Memory Use," page 109, for more information.

Detecting Memory Bottlenecks

The primary symptom of a memory bottleneck is a sustained, high rate of page faults. A **page fault** occurs when an application requests a page, but the system can't find the page at the requested location in RAM.

Two types of page faults can occur:

- **Soft page faults** can occur when the requested page is located elsewhere in RAM. A soft page fault's effect on performance is negligible because disk I/O is not required to find the page.
- **Hard page faults** occur when the requested page is located in virtual memory. The operating system must swap the page out of virtual memory and place it back into RAM. Because of the disk I/O involved, hard page faults slow performance if they occur frequently.

To detect a memory bottleneck, gather information about your system to answer the following questions:

- **How often are requested pages triggering a page fault?** This information gives you an idea of the number of total page faults, both soft and hard page faults, that occur over a period of time.
- **How many pages are retrieved from disk to satisfy page faults?** Compare this information to the preceding information to determine how many hard page faults occur out of the total number of page faults.
- **Does the memory use of any individual application or process climb steadily and never level off?** If so, that application or process is probably leaking memory. In pooled environments, detecting memory leaks is more difficult because pooled connections and prepared statements hold onto memory and can make it appear as if your application is leaking memory even when it isn't. If you run into memory issues when using connection pooling, try tuning the connection pool to reduce the number of connections in the pool. Similarly, try tuning the statement pool to reduce the number of prepared statements in the pool.

For information about tools that can help you troubleshoot memory use, see "The Environment," page 272.

Tuning Your Application and Database Driver to Minimize Memory Use

Here are some general guidelines to minimize memory use:

- **Reduce the number of open connections and prepared statements**—Open connections use memory on the client and on the database server. Make sure that your application closes connections immediately after it's finished with them. If your application uses connection pooling and the database server (or application server) starts to experience memory problems, try tuning the connection pool to reduce the number of connections in the pool. Alternatively, if your database system and database driver supports reauthentication, you may be able to use it to minimize the number of connections required to service your application.

  Using statement pooling with connection pooling complicates the memory situation exponentially. On the database client, client resources that correlate to each pooled statement are stored in memory. On the database server, each pooled connection has a statement pool associated with it that's also maintained in memory. For example, if your application uses 5 pooled connections along with 20 prepared statements, each statement pool associated with those 5 connections may potentially contain all 20 prepared statements. That's 5 connections × 20 prepared statements = 100 prepared statements, all maintained in memory on the database server. If you use statement pooling and the client or database server starts to experience memory problems, try tuning the statement pool to reduce the number of prepared statements in the pool. See "Using Statement Pooling with Connection Pooling," page 238, for more information.
- **Do not leave transactions active for too long**—The database must write every modification made by a transaction to a log that is stored in memory on the database server. If your application uses transactions that update large amounts of data without committing modifications at regular intervals, the application can consume a substantial amount of database memory. Committing a transaction flushes the contents of the log and releases memory used by the database server. See "Managing Commits in Transactions," page 22, for guidelines on committing active transactions.
- **Avoid retrieving large amounts of data from the database server**—When the database driver retrieves data from the database server, it typically stores that data in a result set that is maintained in memory on the client. If your application executes queries that retrieve millions of rows, memory can be used up quickly. Always formulate your SQL queries to retrieve only the data you need.

Similarly, retrieving long data—such as large XML data, long varchar/text, long varbinary, Clobs, and Blobs—can be problematic for memory. Suppose your application executes a query that retrieves hundreds of rows, and those rows happen to contain a Blob. If the database system does not support true LOBs, the database driver will probably emulate this functionality and retrieve the entire Blob across the network and place it in memory on the client. See "Data Retrieval," page 30, for more information.

- **Avoid scrollable cursors unless you know your database system fully supports them**—Scrollable cursors let you go both forward and backward through a result set. Because of limited support for server-side scrollable cursors in many database systems, database drivers often emulate scrollable cursors, storing rows from a scrollable result set in memory on the client or application server. Large scrollable result sets can easily consume memory. See "Using Scrollable Cursors," page 36, for more information.
- **If memory is a limiting factor on your database server, application server, or client, tune your database driver to compensate for that limiting factor**—Some database drivers provide tuning options that allow you to choose how and where some memory-intensive operations are performed. For example, if your client excessively pages to disk because of large result sets, you may want to decrease the size of the fetch buffer, the amount of memory used by the driver to store results retrieved from the database server. Decreasing the fetch buffer size reduces memory consumption, but it means more network round trips, so you need to be aware of the trade-off.

Disk

When an operation reads or writes to disk, performance suffers because disk access is extremely slow. The easiest way to avoid accessing the disk (or disk controller in the case of multiple disks) is to use memory. For example, consider the case of an application that retrieves large result sets. If your client or application server has ample memory and your database driver supports this tuning option, you could increase the size of the fetch buffer on the client to avoid the result set being written to disk. However, remember that if you routinely stretch memory to its limit, paging to disk occurs more frequently. In addition to slowing performance, excessive paging can interfere with other processes that require the same disk, causing disk contention.

Disk contention occurs when multiple processes or threads try to access the same disk simultaneously. The disk limits how many processes/threads can access it and how much data it can transfer. When these limits are reached, processes may have to wait to access the disk. Often, CPU activity is suspended until disk access completes.

If you suspect that disk access occurs more often than it should, the first thing you should do is rule out a memory bottleneck. Once you've ruled out a memory bottleneck, make sure your application avoids unnecessary disk reads and writes so that disk contention rarely happens.

Performance Tip

As a general rule, your application should only access the disk in the following cases: to retrieve database metadata into memory and to write changes to disk, such as in the case of committing transactions.

Table 4-5 lists some common causes for disk bottlenecks and their recommended solutions.

Table 4-5 Causes and Solutions of Disk Bottlenecks

Cause	Solution
Excessive paging caused by a memory bottleneck	Detect and resolve the memory bottleneck. See "Memory," page 107, for more information.
Excessive reads or writes to disk, possibly causing disk contention	Analyze and tune your application to avoid unnecessary disk reads or writes. See "Tuning Your Application to Avoid Unnecessary Disk Reads/Writes," page 112, for more information.

Detecting Disk Bottlenecks

To detect a disk bottleneck, gather information about your system to answer the following questions:

- **Is excessive paging occurring?** A memory bottleneck can resemble a disk bottleneck so it's important to rule out a memory problem before you make any disk improvements. See "Detecting Memory Bottlenecks," page 108, for information about detecting memory bottlenecks.

- **How often is the disk busy?** If your disk has a sustained rate of disk activity of 85% or more for a sustained period of time and a persistent disk queue, you may have a disk bottleneck.

Tuning Your Application to Avoid Unnecessary Disk Reads/Writes

Here are some general guidelines to help your application avoid unnecessary disk reads and writes:

- **Avoid stretching memory to its limit**—Once memory is used up, paging to disk occurs. See "Memory," page 107, for information about detecting and avoiding a memory bottleneck.
- **Avoid using auto-commit mode for transactions**—When using transactions, the database writes every modification made by a transaction to a log that is stored in database memory. A commit tells the database to make those changes permanent. In response, the database writes the changes to disk and flushes the log. In auto-commit mode, transactions are committed automatically by the database, or if the database doesn't support auto-commit mode, by the database driver. You can minimize disk access by using manual commits. See "Managing Commits in Transactions," page 22, for more information.

CPU (Processor)

The CPU is the brain of your database server (or application server), performing most of the calculations and logic required to retrieve data or modify database tables. When the CPU is overly busy, it processes work slowly and may even have jobs waiting in its run queue. When the CPU is too busy to respond to work requests, performance of the database server or application server rapidly hits a ceiling. For example, Figure 4-20 shows benchmark runs of the same driver on different machines with different CPU capacity. As you can see, when run on the machine that is not CPU-bound, performance steadily climbed. On a machine that is CPU bound, performance is capped by the CPU.

Table 4-6 lists some common causes for CPU bottlenecks and their recommended solutions.

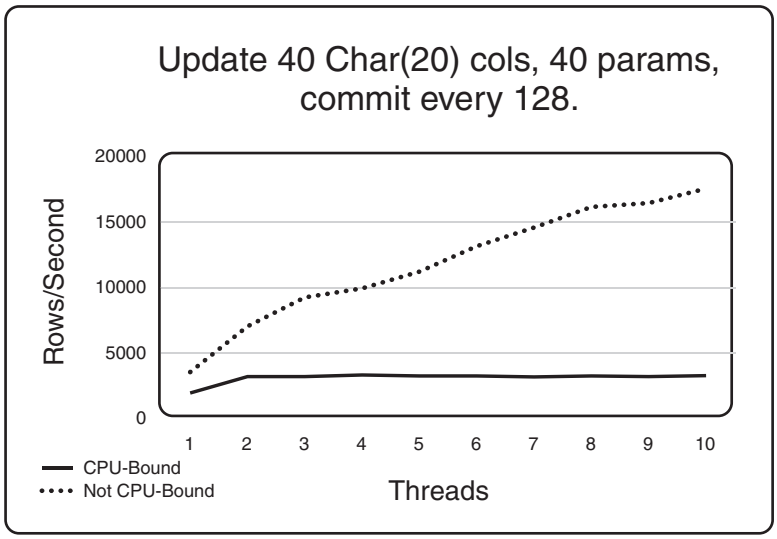


Figure 4-20 CPU-bound versus non-CPU-bound

Table 4-6 Causes and Solutions of CPU Bottlenecks

Cause	Solution
Insufficient CPU capacity	Add multiple processors or upgrade to a more powerful processor.
Inefficient database driver	See "Database Drivers," page 53, for information on why it's important to choose a good database driver.
Poorly optimized application code or database driver	Analyze and tune your application and database driver to minimize CPU use. See "Tuning Your Application or Database Driver to Minimize CPU Use," page 114, for more information.

Detecting CPU Bottlenecks

To detect a CPU bottleneck, gather information about your system to answer the following questions:

- **How much time does the CPU spend executing work?** If the processor is busy 80% or higher for sustained periods, it can be a source of trouble. If you detect

high CPU use, drill down to individual processes to determine if any one application is using more than its fair share of CPU cycles. If so, look more closely at how that application is designed and coded as described in "Tuning Your Application or Database Driver to Minimize CPU Use," page 114.

- **How many processes or threads are waiting to be serviced in the CPU's run queue?** A single queue is used for CPU requests, even on computers with multiple processors. If all processors are busy, threads must wait until CPU cycles are free to perform work. Processes waiting in the queue for sustained periods indicate a CPU bottleneck.
- **What is the rate that processes or threads are switched by the operating system to perform work for other waiting threads?** A **context switch** is the process of storing and restoring the state (context) of a CPU so that multiple processes can share a single CPU resource. Each time the CPU stops running one process and starts running another, a context switch occurs. For example, if your application is waiting for a row lock to release so that it can update data, the operating system may switch the context so that the CPU can perform work on behalf of another application while your application is waiting for the lock to release. Context switching requires significant processor time, so excessive context switches and high CPU use tend to go hand in hand.

For information about tools that can help you troubleshoot CPU use, see Chapter 10, "Troubleshooting Performance Issues."

Tuning Your Application or Database Driver to Minimize CPU Use

Here are some general guidelines to help your application or database driver to minimize CPU use:

- **Maximize query plan reuse**—When a new SQL statement is sent to the database, the database compiles a query plan for that statement and stores it for future reference. Each time a SQL statement is submitted to the database, the database looks for a matching SQL statement and query plan. If a query plan isn't found, the database creates a new query plan for the statement. Each time the database creates a new query plan, it uses CPU cycles. To maximize query plan reuse, consider using statement pooling. For more information about statement pooling, see "Using Statement Pooling," page 236.
- **Ensure connection pools and statement pools are tuned correctly**—Pooling can conserve CPU if tuned correctly, but if not, your pooling envi-

ronment may use more CPU than expected. As a general rule, when the database has to create a new connection or prepared statement, CPU processing becomes expensive. See Chapter 8, "Connection Pooling and Statement Pooling," for information about configuring your pooling environment.

- **Avoid context switching by reducing network round trips**—Each data request that results in a network round trip triggers a context switch. Context switching requires significant processor time, so excessive context switches and high CPU use tend to go hand in hand. Reducing the number of network round trips also reduces the number of context switches. Application design and coding practices that reduce network round trips include connection pooling, statement pooling, avoiding auto-commit mode in favor of manual commits, using local transactions instead of distributed transactions where appropriate, and using batches or arrays of parameters for bulk inserts.
- **Minimize data conversions**—Choose database drivers that convert data efficiently. For example, some database drivers don't support Unicode, a standard encoding that is used for multilingual character sets. If your database driver doesn't support Unicode, more data conversion is required to work with Unicode data, resulting in higher CPU use.

  In addition, choose data types that process efficiently. When you are working with data on a large scale, select the data type that can be processed most efficiently. Retrieving and returning certain data types across the network can increase or decrease network traffic. See "Choosing the Right Data Type," page 34, for details on which data types process more efficiently than others.
- **Be aware that emulated functionality can increase CPU use**—Database drivers sometimes emulate functionality if the database system doesn't support it. While this provides the benefit of interoperability, you should remember that emulated behavior typically uses more CPU because the database driver or the database must perform extra steps to satisfy the behavior. For example, if your application uses scrollable cursors against Oracle, which doesn't support scrollable cursors, CPU use on both the client/application server and database server will be higher than against a database system that does support scrollable cursors, such as DB2. For more information about the type

of functionality that drivers emulate, see "Know Your Database System," page 47.

- **Use data encryption with caution**—Data encryption methods, such as SSL, are CPU-intensive because they require extra steps between the database driver and the database system, to negotiate and agree upon the encryption/decryption information to be used in addition to the process of encrypting the data. To limit the performance penalty associated with data encryption, consider establishing separate connections for encrypted and nonencrypted data. For example, one connection can use encryption for accessing sensitive data such as an individual's tax ID number, while the other connection can forgo encryption for data that is less sensitive, such as an individual's department and title. However, not all database systems allow this. With some database systems, such as Sybase ASE, either all connections to the database use encryption or none of them do. See "Data Encryption across the Network," page 39, for more information.
- **If CPU is a limiting factor on your database server, application server, or client, tune your database driver to compensate for that limiting factor**—Some database drivers provide tuning options that allow you to choose how and where some CPU-intensive operations are performed. For example, Sybase ASE creates stored procedures for prepared statements, a CPU-intensive operation to create the stored procedure, but not to execute it. If your application executes a prepared statement only once, not multiple times, the database server uses more CPU than necessary. Choosing a driver that allows you to tune whether Sybase ASE creates a stored procedure for a prepared statement could improve performance significantly by conserving CPU.

Network Adapter

Computers that are connected to a network have at least one network adapter that sends and receives network packets across the network. Network adapters are designed for a specific network type, such as Ethernet, token-ring, and so on. Differences in the speed of network adapters at either end of the network can cause performance issues. For example, a 64-bit network adapter sends data faster than a 32-bit network adapter can process it.

A sluggish network can indicate that you need more bandwidth, the capacity to send and receive network packets. Increasing bandwidth is similar to widening a congested 2-lane highway into a 4- or 6-lane highway. The highway can handle more traffic, relieving bottlenecks.

Table 4-7 lists some common causes for network bottlenecks and their recommended solutions.

Table 4-7 Causes and Solutions of Network Bottlenecks

Cause	Solution
Insufficient bandwidth	Add more network adapters or upgrade your network adapter. See "Increasing Network Bandwidth," page 106, for more information about upgrading your network adapter. Distribute client connections across multiple network adapters. Reduce network traffic by configuring the database driver to use the maximum database protocol packet size allowed by the database server. See "Configuring Packet Size," page 92, for more information.
Inefficient database driver	See "Database Drivers," page 53, for information on why it's important to choose a good database driver.
Poorly optimized application code or database driver	Analyze and tune your application and database driver to use the network efficiently. See "Tuning Your Application or Database Driver to Use the Network Efficiently," page 118, for more information.

Detecting a Network Bottleneck

What is the rate at which network packets are sent and received using the network adapter? Comparing this rate to the total bandwidth of your network adapter can tell you if the network traffic load is too much for your network adapter. To allow room for spikes in traffic, you should use no more than 50% of capacity.

For information about tools that can help you troubleshoot network use, see "The Environment," page 272.

Tuning Your Application or Database Driver to Use the Network Efficiently

Here are some general guidelines to help your application and database driver use the network efficiently:

- **Reduce the number of network round trips**—Reducing network round trips reduces your application's reliance on the network, which improves performance. Application design and coding practices that reduce network round trips include connection pooling, statement pooling, avoiding auto-commit mode in favor of manual commits, using local transactions instead of distributed transactions where appropriate, and using batches or arrays of parameters for bulk inserts.
- **Tune your database driver to optimize communication across the network**—Some database drivers provide tuning options that allow you to optimize network traffic. For example, if your database driver supports it, you can increase the size of database protocol packets, which ultimately improves performance because it results in fewer network packets being sent across the network. See "Configuring Packet Size," page 92, for more information.
- **Avoid retrieving large amounts of data from the database server**—The more data that must be transferred across the network, the more network packets are required to transfer the data. For example, data such as XML files, Blobs, and Clobs can be very large. Just as retrieving thousands of rows of character data can be a drain on performance, retrieving long data across the network is slow and resource intensive because of the size of the data. Avoid retrieving it unless you have a compelling reason to do so. See "Data Retrieval," page 30, for more information.

  If you can't avoid retrieving data that generates large amounts of network traffic, your application can still control the amount of data being sent from the database by limiting the number of rows sent across the network and reducing the size of each row sent across the network. See "Limiting the Amount of Data Returned," page 34, for more information.
- **Be aware that large result sets can delay your application's response time if you are using a streaming protocol database**—Sybase ASE, Microsoft SQL Server, and MySQL are examples of streaming protocol databases. These database systems process the query and send results until there are no more

results to send; the database is uninterruptable. Therefore, the network connection is "busy" until all results are returned (fetched) to the application. Large result sets can suspend the availability of the network for longer times than small result sets. If you're using a streaming protocol database, it's even more important to reduce the amount of data retrieved from the database server. See "How One Connection for Multiple Statements Works," page 17, for more information about streaming protocol databases versus cursor-based protocol databases.

- **Avoid scrollable cursors unless you know your database system fully supports them**—Scrollable cursors provide the ability to go both forward and backward through a result set. Because of limited support for server-side scrollable cursors in many database systems, database drivers often emulate scrollable cursors, storing rows from a scrollable result set in memory on the client or application server. A large result set can result in large amounts of data being transferred over the network. See "Using Scrollable Cursors," page 36, for more information.

Virtualization

You may have heard talk about a recent trend in database computing known as virtualization. **Virtualization** allows companies to consolidate server resources by allowing multiple operating system instances to run at the same time on a single physical computer. A single server can run 4, 8, 16, or even more virtual operating systems. In 2007, the number of companies offering virtualization management solutions increased from 6 to 50, a staggering 866% increase. It's not hard to figure out why.

Over the past 10 years, hardware has become less expensive and more powerful. To keep pace with computing demands, companies have acquired large numbers of server machines, but they often find themselves low on the space, power, and air conditioning required to store and maintain them. It's estimated that in an unvirtualized environment, only 8% to 10% of the capacity of a server is used. Using virtualization, companies can get more work out of fewer machines, easing and sometimes eliminating the costs associated with housing multiple servers. For example, imagine an IT's data center that maintains 50 servers stored in a crowded, subleased server space. By creating 5 virtual servers on only 10 servers,

that data center can move to a smaller space and get rid of an expensive sublease without sacrificing business capability.

What does virtualization mean to the performance of database applications? First, it's important to choose a database driver that supports virtualization technologies. Next, you need to be aware that it becomes easier to stretch hardware resources such as network, memory, CPU, and disk use to their limits; the probability that your database application will be affected by performance issues caused by hardware constraints is amplified.

Finally, it's important to understand that virtualized environments make it harder to detect where performance bottlenecks actually originate because of the increased complexity of the environment. In addition, there's no overarching tool that is operating system-agnostic to analyze resource use in virtualized environments (although companies are rushing to develop virtualization management tools that allow you to monitor activity and resource use). For example, Figure 4-21 shows a virtualized machine that runs four operating systems and hosts four applications. If Application A and Application C routinely generate a spike in network traffic at the same time every day, the network adapter of the virtualized machine may not be able to handle the increase in network requests. The increase can affect the performance of not only Application A and C, but also the performance of Application B and D.

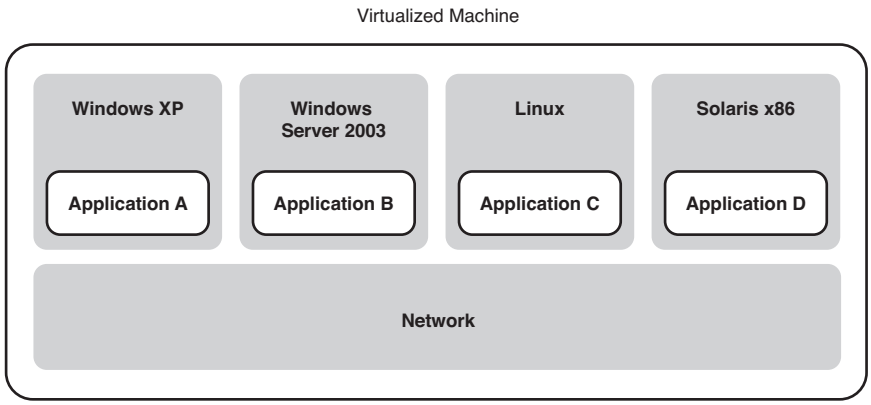


Figure 4-21 Virtualized machine running multiple operating systems and database applications

The soundest advice we can give is to invest in the best hardware you can afford. Software tools to help you troubleshoot performance issues in virtualized environments demand a steep learning curve and, ultimately, will cost more than the best hardware. In the example shown in Figure 4-21, if we provide each virtualized machine with its own dedicated network adapter, our performance bottleneck is resolved.

In a few cases, it may not be feasible to add or upgrade hardware resources to expand those that are being overworked. For example, each computer has a physical limit on the amount of memory it can address. When hardware becomes the limiting factor for performance, using an efficient database driver becomes even more important. See "Database Drivers," page 53, for more information about choosing an efficient database driver.

Summary

The environment in which your database application runs affects its performance. In the Java environment, performance can vary between JVMs from different manufacturers, so it's important to choose a JVM that gives your application the best performance. You can further improve performance by tuning JVM settings for heap size and garbage collection. In contrast, the .NET CLR doesn't provide the same tuning ability for garbage collection, and efficient garbage collection largely depends on your application code.

Any operating system change, even a minor one, can affect performance more than you would think. One often overlooked factor is the endianness of the operating system, as determined by the computer's processor. If possible, try to align the endianness of the operating system on the database client with that of the operating system on the database server.

Database clients and database servers communicate over a network, typically TCP/IP. How efficiently your database application uses that network affects performance. Following are key techniques for ensuring the best performance over the network:

- Reducing network round trips
- Tuning the size of database protocol packets
- Reducing the number of network hops between network destinations
- Avoiding network packet fragmentation

Hardware resources such as memory, disk I/O, CPU, and the network adapter can be a limiting factor for performance. To conserve hardware resources, you often can tune the database driver or use specific application design and coding techniques. In virtualized environments, it's easier to stretch hardware resources to their limits and harder to detect where bottlenecks originate. Investing in the best hardware that you can afford will save you in the long run.