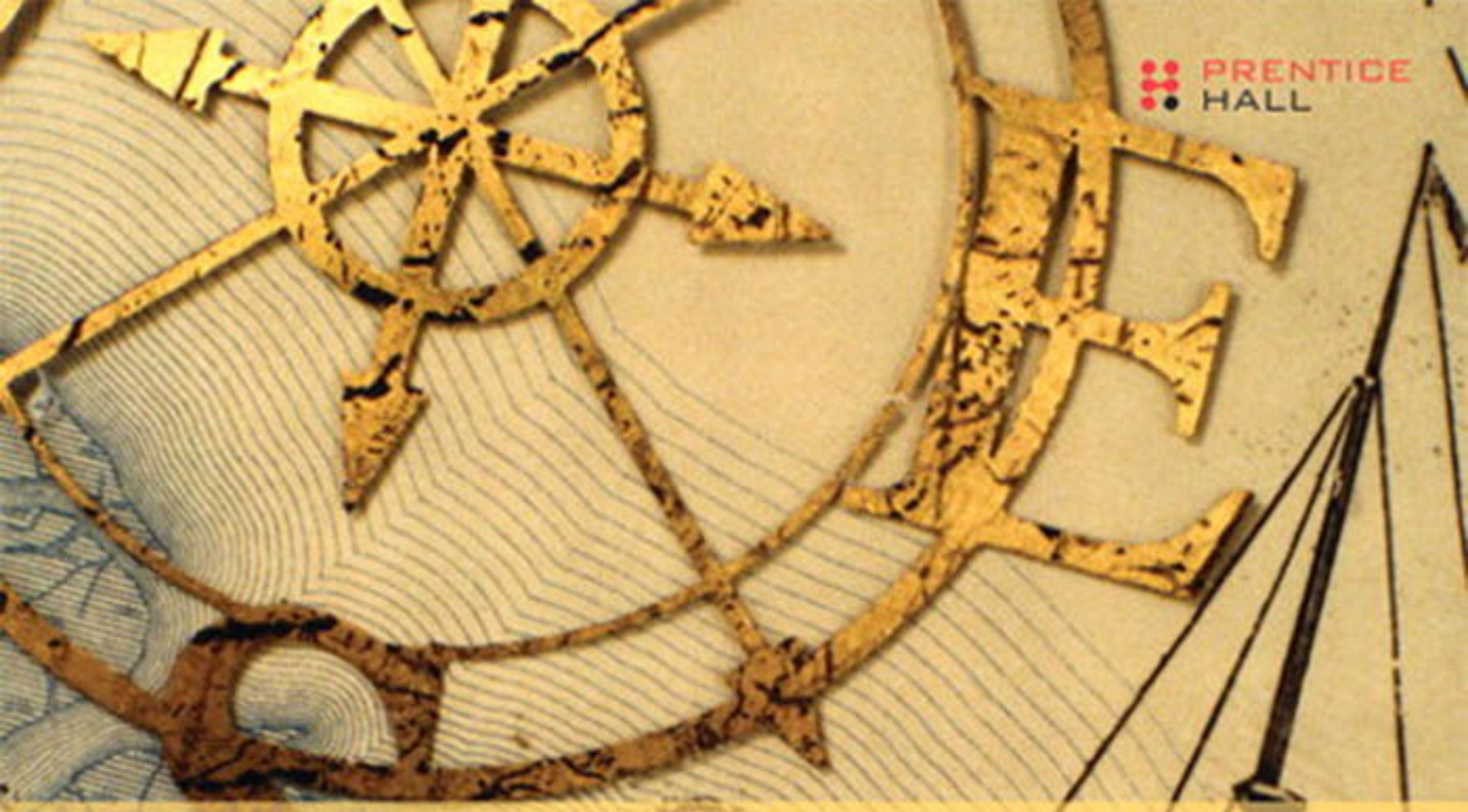

PRENTICE HALL

THE DATA ACCESS HANDBOOK

Achieving Optimal Database Application Performance and Scalability

JOHN GOODSON • ROBERT A. STEWARD

Library of Congress Cataloging-in-Publication Data
Goodson, John, 1964-
The data access handbook : achieving optimal database application performance and scalability / John Goodson and Robert A. Steward. — 1st ed.
p. cm.
ISBN 978-0-13-714393-1 (pbk. : alk. paper) 1. Database design—Handbooks, manuals, etc. 2. Application software—Development—Handbooks, manuals, etc. 3. Computer networks—Handbooks, manuals, etc. 4. Middleware—Handbooks, manuals, etc. I. Steward, Robert A. (Robert Allan) II. Title.
QA76.9.D26G665 2009
005.3—dc22
2008054864

ISBN-13: 978-0-137-14393-1
ISBN-10: 0-137-14393-1
Text printed in the United States on recycled paper at RR Donnelley, Crawfordsville, Indiana.
First printing March 2009

CHAPTER TWO

Designing for Performance: What's Your Strategy?

Designing your database application and the configuration of the database middleware that connects your application to the database server for optimal performance isn't easy. We refer to all these components as your database application deployment. There is no one-size-fits-all design. You must think about every component to get the best performance possible.

Often you are not in control of every component that affects performance. For example, your company may dictate that all applications run on an application server. Also, your database administrator most likely controls the database server machine's configuration. In these cases, you need to consider the configurations that are dictated when designing your database application deployment. For example, if you know that the applications will reside on an application server, you probably want to spend ample time planning for connection and statement pooling, which are both discussed in this chapter.

Your Applications

Many software architects and developers don't think that the design of their database applications impacts the performance of those applications. This is not true; application design is a key factor. An application is often coded to establish a new connection to gather information about the database, such as supported data types or database version. Avoid establishing additional connections for this purpose because connections are performance-expensive, as we explain in this chapter.

This section explores several key functional areas of applications that you need to consider to achieve maximum performance:

- Database connections
- Transactions
- SQL statements
- Data retrieval

Some functional areas of applications, such as data encryption, affect performance, but you can do little about the performance impact. We discuss these areas and provide information about the performance impact you can expect.

When you make good application design decisions, you can improve performance by doing the following:

- Reducing network traffic
- Limiting disk I/O
- Optimizing application-to-driver interaction
- Simplifying queries

For API-specific code examples and discussions, you should also read the chapter for the standards-based API that you work with:

- For ODBC users, see Chapter 5, "ODBC Applications: Writing Good Code."
- For JDBC users, see Chapter 6, "JDBC Applications: Writing Good Code."
- For ADO.NET users, see Chapter 7, ".NET Applications: Writing Good Code."

Database Connections

The way you implement database connections may be the most important design decision you make for your application.

Your choices for implementing connections are as follows:

- Obtain a connection from a connection pool. Read the section, "Using Connection Pooling," page 12.
- Create a new connection one at a time as needed. Read the section, "Creating a New Connection One at a Time as Needed," page 16.

The right choice mainly depends on the CPU and memory conditions on the database server, as we explain throughout this section.

Facts About Connections

Before we discuss how to make this decision, here are some important facts about connections:

- Creating a connection is performance-expensive compared to all other tasks a database application can perform.
- Open connections use a substantial amount of memory on both the database server and database client machines.
- Establishing a connection takes multiple network round trips to and from the database server.
- Opening numerous connections can contribute to out-of-memory conditions, which might cause paging of memory to disk and, thus, overall performance degradation.
- In today's architectures, many applications are deployed in connection pooled environments, which are intended to improve performance. However, many times poorly tuned connection pooling can result in performance degradation. Connection pools can be difficult to design, tune, and monitor.

Why Connections Are Performance-Expensive

Developers often assume that establishing a connection is a simple request that results in the driver making a single network round trip to the database server to initialize a user. In reality, a connection typically involves many network round trips between the driver and the database server. For example, when a driver connects to Oracle or Sybase, that connection may take anywhere from seven to ten network round trips to perform the following actions:

- Validate the user's credentials.
- Negotiate code page settings between what the database driver expects and what the database has available, if necessary.

- Get database version information.
- Establish the optimal database protocol packet size to be used for communication.
- Set session settings.

In addition, the database management system establishes resources on behalf of the connection, which involves performance-expensive disk I/O and memory allocation.

You might be thinking that you can eliminate network round trips if you place your applications on the same machine as the database system. This is, in most cases, not realistic because of the complexity of real-world enterprises—many, many applications accessing many database systems with applications running on several application servers. In addition, the server on which the database system runs must be well tuned for the database system, not for many different applications. Even if one machine would fit the bill, would you really want a single point of failure?

Using Connection Pooling

A **connection pool** is a cache of physical database connections that one or more applications can reuse. Connection pooling can provide significant performance gains because reusing a connection reduces the overhead associated with establishing a physical connection. The caveat here is that your database server must have enough memory to manage all the connections in the pool.

In this book, we discuss client-side connection pooling (connection pooling provided by database drivers and application servers), not database-side connection pooling (connection pooling provided by database management systems). Some database management systems provide connection pooling, and those implementations work in conjunction with client-side connection pooling. Although specific characteristics of database-side connection pooling vary, the overall goal is to eliminate the overhead on the database server of establishing and removing connections. Unlike client-side connection pooling, database-side connection pooling does not optimize network round trips to the application. As we stated previously, connecting to a database is performance-expensive because of the resource allocation in the database driver (network round trips between the driver and the database), and the resource allocation on the database server. Client-side connection pooling helps solve the issue of expensive resource allocation for both the database driver and database server. Database-side connection pooling only helps solve the issue on the database server.

How Connection Pooling Works

In a pooled environment, once the initial physical connection is established, it will likely not be closed for the life of the environment. That is, when an application disconnects, the physical connection is not closed; instead, it is placed in the pool for reuse. Therefore, re-establishing the connection becomes one of the fastest operations instead of one of the slowest.

Here is a basic overview of how connection pooling works (as shown in Figure 2-1):

1. When the application or application server is started, the connection pool is typically populated with connections.
2. An application makes a connection request.
3. Either the driver or the Connection Pool Manager (depending on your architecture) assigns one of the pooled connections to the application instead of requesting that a new connection be established. This means that no network round trips occur between the driver and the database server for connection requests because a connection is available in the pool. The result: Your connection request is *fast*.
4. The application is connected to the database.
5. When the connection is closed, it is placed back into the pool.

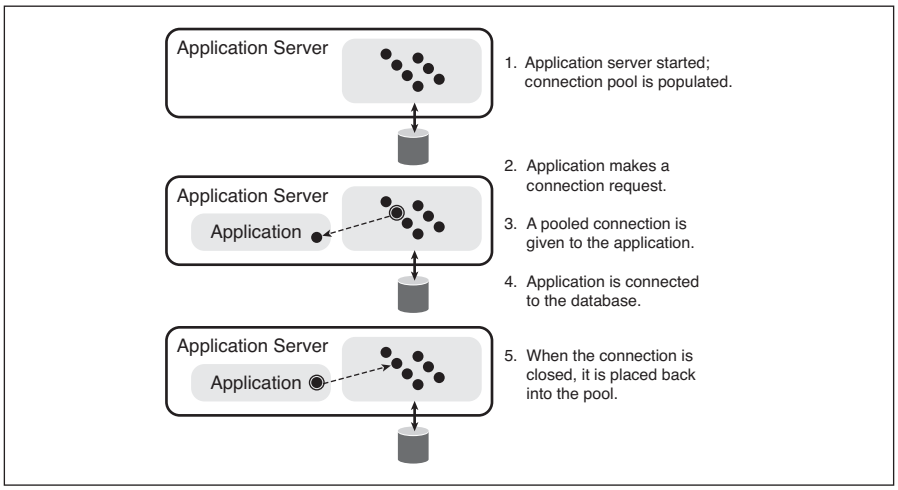

Figure 2-1 Connection pooling

Guidelines for Connection Pooling

Here are some general guidelines for using connection pooling. For details about different connection pooling models, see Chapter 8, "Connection Pooling and Statement Pooling."

- A perfect scenario for using connection pooling is when your applications reside on an application server, which implies multiple users using the applications.
- Consider using connection pooling if your application has multiple users and your database server has enough memory to manage the maximum number of connections that will be in the pool at any given time. In most connection pooling models, it is easy to calculate the maximum number of connections that will be in a pool because the connection pool implementation allows you to configure the maximum. If the implementation you are using does not support configuring the maximum number of connections in a pool, you must calculate how many connections will be in the pool during peak times to determine if your database server can handle the load.
- Determine whether the number of database licenses you have accommodates a connection pool. If you have limited licenses, answer the following questions to determine if you have enough licenses to support a connection pool:
    a. Will other applications use database licenses? If yes, take this into account when calculating how many licenses you need for your connection pool.
    b. Are you using a database that uses a streaming protocol, such as Sybase, Microsoft SQL Server, or MySQL? If yes, you may be using more database connections than you think. In streaming protocol databases, only one request can be processed at a time over a single connection; the other requests on the same connection must wait for the preceding request to complete before a subsequent request can be processed. Therefore, some database driver implementations duplicate connections (establish another connection) when multiple requests are sent over a single connection so that all requests can be processed in a timely manner.
- When you develop your application to use connection pooling, open connections just before the application needs them. Opening them earlier than

necessary decreases the number of connections available to other users and can increase the demand for resources. Don't forget to close them when the database work is complete so that the connection can return to the pool for reuse.

When Not to Use Connection Pooling

Some applications are not good candidates for using connection pooling. If your applications have any of the following characteristics, you probably don't want to use connection pooling. In fact, for these types of applications, connection pooling may degrade performance.

- **Applications that restart numerous times daily**—This typically applies only to architectures that are not using an application server. Depending on the configuration of the connection pool, it may be populated with connections each time the application is started, which causes a performance penalty up front.
- **Single-user applications, such as report writers**—If your application only needs to establish a connection for a single user who runs reports two to three times daily, the memory usage on the database server associated with a connection pool degrades performance more than establishing the connection two or three times daily.
- **Applications that run single-user batch jobs, such as end-of-day/week/month reporting**—Connection pooling provides no advantage for batch jobs that access only one database server, which typically equates to only one connection. Furthermore, batch jobs are usually run during off hours when performance is not as much of a concern.

Performance Tip

When your application does not use connection pooling, avoid connecting and disconnecting multiple times throughout your application to execute SQL statements, because each connection might use five to ten times as many network requests as the SQL statement.

Creating a New Connection One at a Time as Needed

When you create a new connection one at a time as needed, you can design your application to create either of the following:

- One connection for each statement to be executed
- One connection for multiple statements, which is often referred to as using multiple threads

Figure 2-2 compares these two connection models.

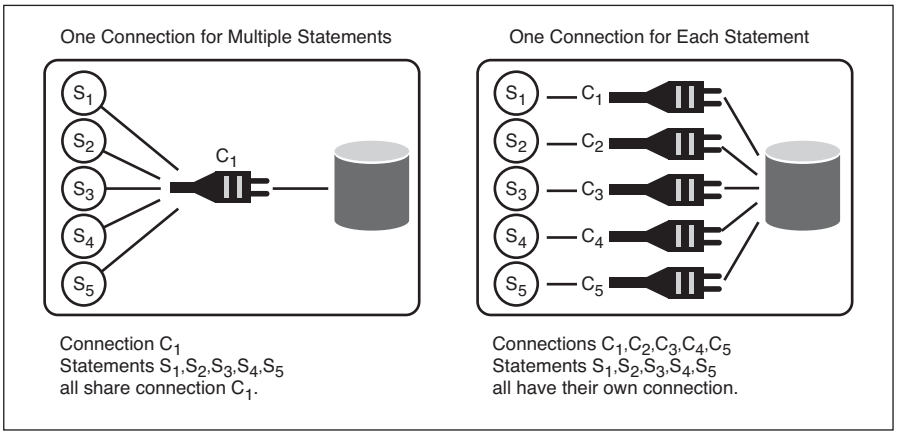

Figure 2-2 Comparing two connection models

The advantage of using one connection for each statement is that each statement can access the database at the same time. The disadvantage is the overhead of establishing multiple connections.

The advantages and disadvantages of using one connection for multiple statements are explained later in this section.

One Connection for Multiple Statements

Before we can explain the details of one connection for multiple statements, we need to define **statement**. Some people equate "statement" to "SQL statement." We like the definition of "statement" that is found in the *Microsoft ODBC 3.0 Programmer's Reference*:

> *A statement is most easily thought of as an SQL statement, such as* `SELECT * FROM Employee`. *However, a statement is more than just an SQL statement—it consists of all of the information associated with that SQL statement, such as any result sets created by the statement and parameters used in the execution of the statement. A statement does not even need to have an application-defined SQL statement. For example, when a catalog function such as* `SQLTables` *is executed on a statement, it executes a predefined SQL statement that returns a list of table names.*[1]

To summarize, a statement is not only the request sent to the database but the result of the request.

How One Connection for Multiple Statements Works

Note

Because of the architecture of the ADO.NET API, this connection model typically does not apply.

When you develop your application to use one connection for multiple statements, an application may have to wait for a connection. To understand why, you must understand how one connection for multiple statements works; this depends on the protocol of the database system you are using: streaming or cursor based. Sybase, Microsoft SQL Server, and MySQL are examples of streaming protocol databases. Oracle and DB2 are examples of cursor-based protocol databases.

Streaming protocol 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.

Cursor-based protocol database systems assign a database server-side "name" (cursor) to a SQL statement. The server operates on that cursor in incremental time segments. The driver tells the database server when to work and how much information to return. Several cursors can use the network connection, each working in small slices of time.

[1] *Microsoft ODBC 3.0 Programmer's Reference and SDK Guide*, Volume I. Redmond: Microsoft Press, 1997

Example A: Streaming Protocol Result Sets

Let's look at the case where your SQL statement creates result sets and your application is accessing a streaming protocol database. In this case, the connection is unavailable to process another SQL statement until the first statement is executed and all results are returned to the application. The time this takes depends on the size of the result set. Figure 2-3 shows an example.

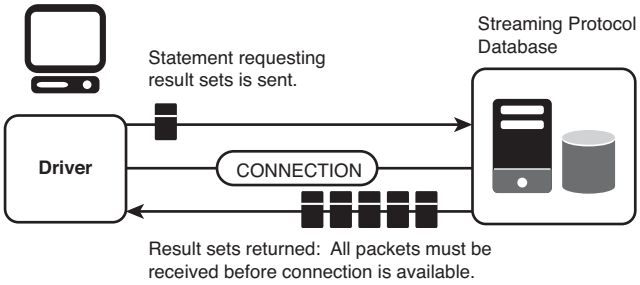

Figure 2-3 Streaming protocol result sets

Example B: Streaming Protocol Updates

Let's look at the case where the SQL statement updates the database and your application is accessing a streaming protocol database, as shown in Figure 2-4. The connection is available as soon as the statement is executed and the row count is returned to the application.

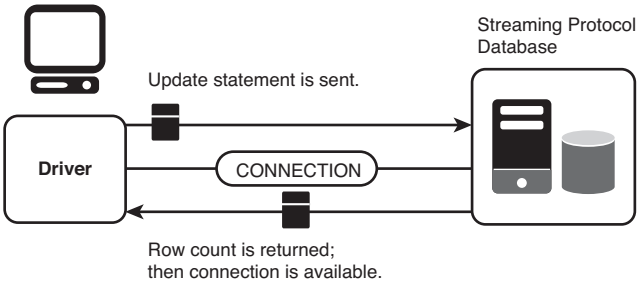

Figure 2-4 Streaming protocol updates

Example C: Cursor-Based Protocol/Result Sets

Last, let's look at the case where your SQL statement creates result sets and your application is accessing a cursor-based protocol database. Unlike Example A, which is a streaming protocol example, the connection is available before all the results are returned to the application. When using cursor-based protocol databases, the result sets are returned as the driver asks for them. Figure 2-5 shows an example.

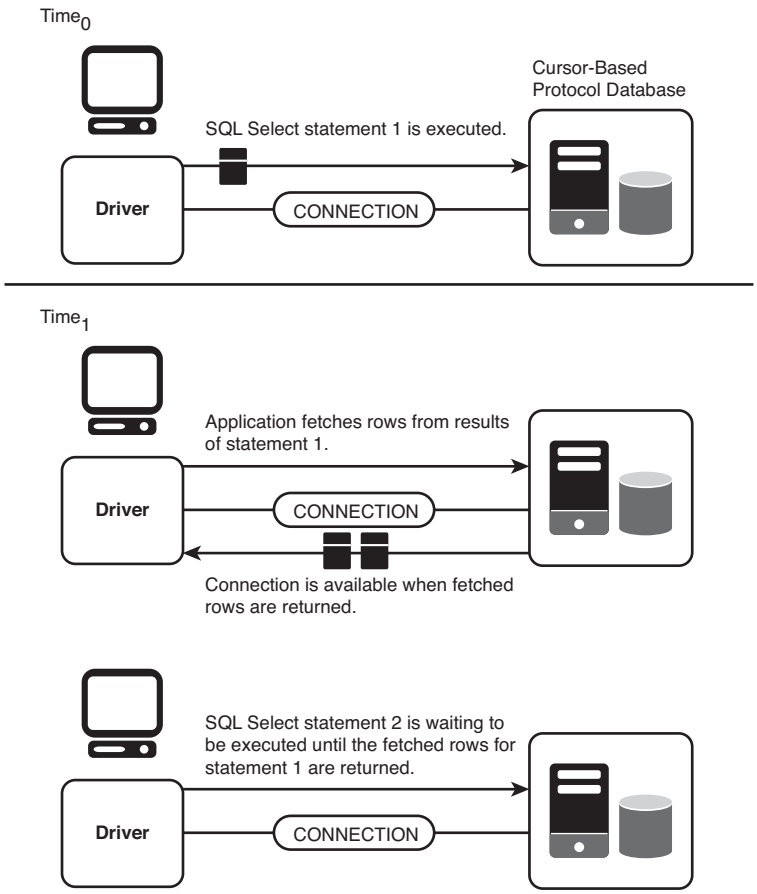

Figure 2-5 Cursor-based protocol/result sets

Advantages and Disadvantages

The advantage of using one connection for multiple statements is that it reduces the overhead of establishing multiple connections, while allowing multiple statements to access the database. The overhead is reduced on both the database server and client machines.

The disadvantage of using this method of connection management is that the application may have to wait to execute a statement until the single connection is available. We explained why in "How One Connection for Multiple Statements Works," page 17.

Guidelines for One Connection for Multiple Statements

Here are some guidelines for when to use one connection for multiple statements:

- Consider using this connection model when your database server has hardware constraints such as limited memory and one or more of the following conditions are true:
  a. You are using a cursor-based protocol database.
  b. The statements in your application return small result sets or no result sets.
  c. Waiting for a connection is acceptable. The amount of time that is acceptable for the connection to be unavailable depends on the requirements of your application. For example, 5 seconds may be acceptable for an internal application that logs an employee's time but may not be acceptable for an online transaction processing (OLTP) application such as an ATM application. What is an acceptable response time for your application?
- This connection model should not be used when your application uses transactions.

Case Study: Designing Connections

Let's look at one case study to help you understand how to design database connections. The environment details are as follows:

- The environment includes a middle tier that must support 20 to 100 concurrent database users, and performance is key.

- CPU and memory are plentiful on both the middle tier and database server.
- The database is Oracle, Microsoft SQL Server, Sybase, or DB2.
- The API that the application uses is ODBC, JDBC, or ADO.NET.
- There are 25 licenses for connections to the database server.

Here are some possible solutions:

- Solution 1: Use a connection pool with a maximum of 20 connections, each with a single statement.
- Solution 2: Use a connection pool with a maximum of 5 connections, each with 5 statements.
- Solution 3: Use a single connection with 5 to 25 statements.

The key information in this case study is the ample CPU and memory on both the middle tier and database server and the ample number of licenses to the database server. The other information is really irrelevant to the design of the database connections.

Solution 1 is the best solution because it performs better than the other two solutions. Why? Processing one statement per connection provides faster results for users because all the statements can access the database at the same time.

The architecture for Solutions 2 and 3 is one connection for multiple statements. In these solutions, the single connection can become a bottleneck, which means slower results for users. Therefore, these solutions do not meet the requirement of "performance is key."

Transaction Management

A **transaction** is one or more SQL statements that make up a unit of work performed against the database, and either all the statements in a transaction are committed as a unit or all the statements are rolled back as a unit. This unit of work typically satisfies a user request and ensures data integrity. For example, when you use a computer to transfer money from one bank account to another, the request involves a transaction: updating values stored in the database for both accounts. For a transaction to be completed and database changes to be made permanent, a transaction must be completed in its entirety.

What is the correct transaction commit mode to use in your application? What is the right transaction model for your database application: local or distributed? Use the guidelines in this section to help you manage transactions more efficiently.

You should also read the chapter for the standards-based API that you work with; these chapters provide specific examples for each API:

- For ODBC users, see Chapter 5.
- For JDBC users, see Chapter 6.
- For ADO.NET users, see Chapter 7.

Managing Commits in Transactions

Committing (and rolling back) transactions is slow because of the disk I/O and potentially the number of network round trips required. What does a commit actually involve? The database must write to disk every modification made by a transaction to the database. This is usually a sequential write to a journal file (or log); nevertheless, it involves expensive disk I/O.

In most standards-based APIs, the default transaction commit mode is auto-commit. In auto-commit mode, a commit is performed for every SQL statement that requires a request to the database, such as `Insert`, `Update`, `Delete`, and `Select` statements. When auto-commit mode is used, the application does not control when database work is committed. In fact, commits commonly occur when there's actually no real work to commit.

Some database systems, such as DB2, do not support auto-commit mode. For these databases, the database driver, by default, sends a commit request to the database after every successful operation (SQL statement). This request equates to a network round trip between the driver and the database. The round trip to the database occurs even though the application did not request the commit and even if the operation made no changes to the database. For example, the driver makes a network round trip even when a `Select` statement is executed.

Because of the significant amount of disk I/O required to commit every operation on the database server and because of the extra network round trips that occur between the driver and the database, in most cases you will want to turn off auto-commit mode in your application. By doing this, your application can control when the database work is committed, which provides dramatically better performance.

Consider the following real-world example. ASoft Corporation coded a standards-based database application and experienced poor performance in testing. Its performance analysis showed that the problem resided in the bulk five million `Insert` statements sent to the database. With auto-commit mode on, this meant an additional five million `Commit` statements were being issued across the

network and that every inserted row was written to disk immediately following the execution of the `Insert`. When auto-commit mode was turned off in the application, the number of statements issued by the driver and executed on the database server was reduced from ten million (five million `Inserts` + five million `Commits`) to five million and one (five million `Inserts` + one `Commit`). As a consequence, application processing was reduced from eight hours to ten minutes. Why such a dramatic difference in time? There was significantly less disk I/O required by the database server, and there were 50% fewer network round trips.

Performance Tip

Although turning off auto-commit mode can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for longer than necessary, preventing other users from accessing the rows. Typically, committing transactions in intervals provides the best performance as well as acceptable concurrency.

If you have turned off auto-commit mode and are using manual commits, when does it make sense to commit work? It depends on the following factors:

- The type of transactions your application performs. For example, does your application perform transactions that modify or read data? If your application modifies data, does it update large amounts of data?
- How often your application performs transactions.

For most applications, it's best to commit a transaction after every logical unit of work. For example, consider a banking application that allows users to transfer money from one account to another. To protect the data integrity of that work, it makes sense to commit the transaction after both accounts are updated with the new amounts.

However, what if an application allows users to generate reports of account balances for each day over a period of months? The unit of work is a series of `Select` statements, one executed after the other to return a column of balances. In most cases, for every `Select` statement executed against the database, a lock is placed on rows to prevent another user from updating that data. By holding

locks on rows for longer than necessary, active transactions can prevent other users from updating data, which ultimately can reduce throughput and cause concurrency issues. In this case, you may want to commit the `Select` statements in intervals (after every five `Select` statements, for example) so that locks are released in a timely manner.

In addition, be aware that leaving transactions active consumes database memory. Remember that the database must write every modification made by a transaction to a log that is stored in database memory. Committing a transaction flushes the contents of the log and releases database memory. If your application uses transactions that update large amounts of data (1,000 rows, for example) without committing modifications, the application can consume a substantial amount of database memory. In this case, you may want to commit after every statement that updates a large amount of data.

How often your application performs transactions also determines when you should commit them. For example, if your application performs only three transactions over the course of a day, commit after every transaction. In contrast, if your application constantly performs transactions that are composed of `Select` statements, you may want to commit after every five `Select` statements.

Isolation Levels

We will not go into the details of isolation levels in this book, but architects should know the default transaction isolation level of the database system they are using. A **transaction isolation level** represents a particular locking strategy used in the database system to improve data integrity.

Most database systems support several isolation levels, and the standards-based APIs provide ways for you to set isolation levels. However, if the database driver you are using does not support the isolation level you set in your application, the setting has no effect. Make sure you choose a driver that gives you the level of data integrity that you need.

Local Transactions Versus Distributed Transactions

A **local transaction** is a transaction that accesses and updates data on only one database. Local transactions are significantly faster than distributed transactions because local transactions do not require communication between multiple databases, which means less logging and fewer network round trips are required to perform local transactions.

Use local transactions when your application does *not* have to access or update data on multiple networked databases.

A **distributed transaction** is a transaction that accesses and updates data on multiple networked databases or systems and must be coordinated among those databases or systems. These databases may be of several types located on a single server, such as Oracle, Microsoft SQL Server, and Sybase; or they may include several instances of a single type of database residing on numerous servers.

The main reason to use distributed transactions is when you need to make sure that databases stay consistent with one another. For example, suppose a catalog company has a central database that stores inventory for all its distribution centers. In addition, the company has a database for its east coast distribution center and one for the west coast. When a catalog order is placed, an application updates the central database and updates either the east or west coast database. The application performs both operations in one distributed transaction to ensure that the information in the central database remains consistent with the information in the appropriate distribution center's database. If the network connection fails before the application updates both databases, the entire transaction is rolled back; neither database is updated.

Distributed transactions are substantially slower than local transactions because of the logging and network round trips needed to communicate between all the components involved in the distributed transaction.

For example, Figure 2-6 shows what happens during a local transaction.

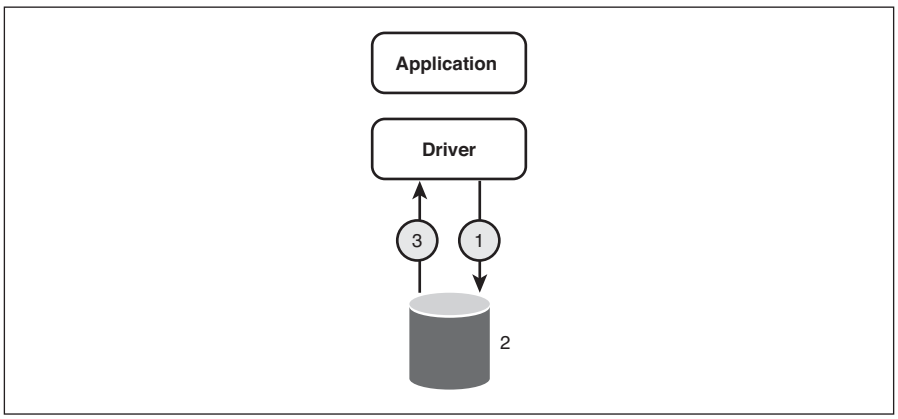

Figure 2-6 Local transaction

The following occurs when the application requests a transaction:

1. The driver issues a commit request.
2. If the database can commit the transaction, it does, and writes an entry to its log. If it cannot, it rolls back the transaction.
3. The database replies with a status to the driver indicating if the commit succeeded or failed.

Figure 2-7 shows what happens during a distributed transaction, in which all databases involved in the transaction must either commit or roll back the transaction.

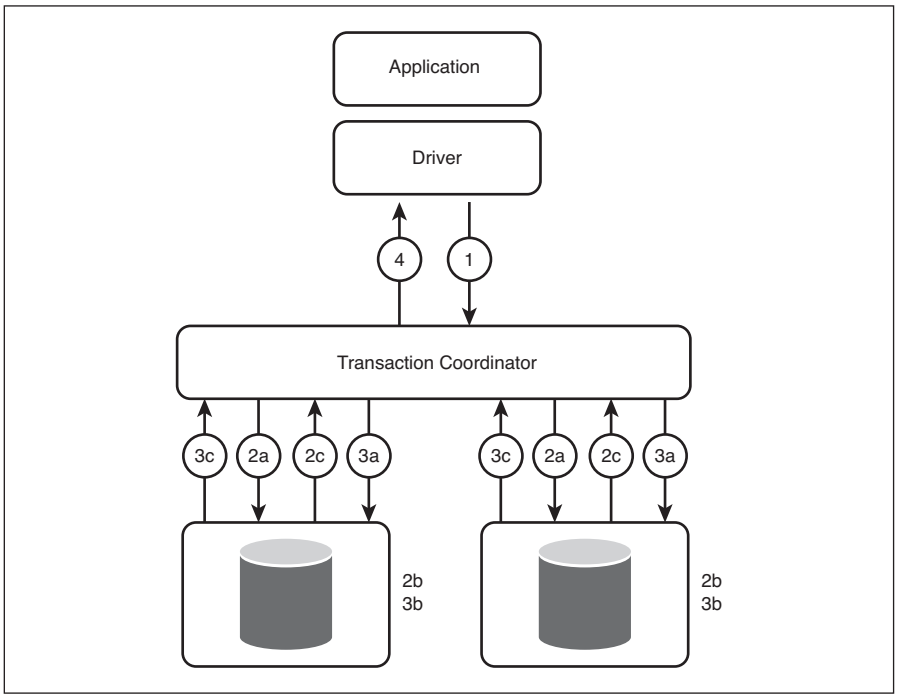

Figure 2-7 Distributed transaction

The following occurs when the application requests a transaction:

1. The driver issues a commit request.
2. The transaction coordinator sends a precommit request to all databases involved in the transaction.

   a. The transaction coordinator sends a commit request command to all databases.
   b. Each database executes the transaction up to the point where the database is asked to commit, and each writes recovery information to its logs.
   c. Each database replies with a status message to the transaction coordinator indicating whether the transaction up to this point succeeded or failed.
3. The transaction coordinator waits until it has received a status message from each database. If the transaction coordinator received a status message from all databases indicating success, the following occurs:
   a. The transaction coordinator sends a commit message to all the databases.
   b. Each database completes the commit operation and releases all the locks and resources held during the transaction.
   c. Each database replies with a status to the transaction coordinator indicating whether the operation succeeded or failed.
4. The transaction coordinator completes the transaction when all acknowledgments have been received and replies with a status to the driver indicating if the commit succeeded or failed.

Note for Java Users

The default transaction behavior of many Java application servers uses distributed transactions, so changing that default transaction behavior to local transactions, if distributed transactions are not required, can improve performance.

SQL Statements

Will your application have a defined set of SQL statements that are executed multiple times? If your answer is yes, you will most likely want to use prepared statements and statement pooling if your environment supports it.

Using Statements Versus Prepared Statements

A **prepared statement** is a SQL statement that has been compiled, or prepared, into an access or query plan for efficiency. A prepared statement is available for reuse by the application without the overhead in the database of re-creating the query plan. A prepared statement is associated with one connection and is available until it is explicitly closed or the owning connection is closed.

Most applications have a set of SQL statements that are executed multiple times and a few SQL statements that are executed only once or twice during the life of an application. Although the overhead for the initial execution of a prepared statement is high, the advantage is realized with subsequent executions of the SQL statement. To understand why, let's examine how a database processes a SQL statement.

The following occurs when a database receives a SQL statement:

1. The database parses the statement and looks for syntax errors.
2. The database validates the user to make sure the user has privileges to execute the statement.
3. The database validates the semantics of the statement.
4. The database figures out the most efficient way to execute the statement and prepares a query plan. Once the query plan is created, the database can execute the statement.

When a prepared query is sent to the database, the database saves the query plan until the driver closes it. This allows the query to be executed time and time again without repeating the steps described previously. For example, if you send the following SQL statement to the database as a prepared statement, the database saves the query plan:

```
SELECT * FROM Employees WHERE SSID = ?
```

Note that this SQL statement uses a parameter marker, which allows the value in the `WHERE` clause to change for each execution of the statement. Do not use a literal in a prepared statement unless the statement will be executed with the same value(s) every time. This scenario would be rare.

Using a prepared statement typically results in at least two network round trips to the database server:

- One network round trip to parse and optimize the query
- One or more network round trips to execute the query and retrieve the results

Performance Tip

If your application makes a request only once during its life span, it is better to use a statement than a prepared statement because it results in only a single network round trip. Remember, reducing network communication typically provides the most performance gain. For example, if you have an application that runs an end-of-day sales report, the query that generates the data for that report should be sent to the database server as a statement, not as a prepared statement.

Note that not all database systems support prepared statements; Oracle, DB2, and MySQL do, and Sybase and Microsoft SQL Server do not. If your application sends prepared statements to either Sybase or Microsoft SQL Server, these database systems create stored procedures. Therefore, the performance of using prepared statements with these two database systems is slower.

Some database systems, such as Oracle and DB2, let you perform a prepare and execute together. This functionality provides two benefits. First, it eliminates a round trip to the database server. Second, when designing your application, you don't need to know whether you plan to execute the statement again, which allows you to optimize the next execution of the statement automatically.

Read the next section about statement pooling to see how prepared statements and statement pooling go hand in hand.

Statement Pooling

If you have an application that repeatedly executes the same SQL statements, statement pooling can improve performance because it prevents the overhead of repeatedly parsing and creating cursors (server-side resource to manage the SQL request) for the same statement, along with the associated network round trips.

A **statement pool** is a group of prepared statements that an application can reuse. Statement pooling is not a feature of a database system; it is a feature of database drivers and application servers. A statement pool is owned by a physical connection, and prepared statements are placed in the pool after their initial execution. For details about statement pooling, see Chapter 8, "Connection Pooling and Statement Pooling."

How does using statement pooling affect whether you use a statement or a prepared statement?

- If you are using statement pooling and a SQL statement will only be executed once, use a statement, which is not placed in the statement pool. This avoids the overhead associated with finding that statement in the pool.
- If a SQL statement will be executed infrequently but may be executed multiple times during the life of a statement pool, use a prepared statement. Under similar circumstances without statement pooling, use a statement. For example, if you have some statements that are executed every 30 minutes or so (infrequently), the statement pool is configured for a maximum of 200 statements, and the pool never gets full, use a prepared statement.

Data Retrieval

To retrieve data efficiently, do the following:

- Return only the data you need. Read "Retrieving Long Data," page 31.
- Choose the most efficient way to return the data. Read "Limiting the Amount of Data Returned," page 34, and "Choosing the Right Data Type," page 34.
- Avoid scrolling through the data. Read "Using Scrollable Cursors," page 36.
- Tune your database middleware to reduce the amount of information that is communicated between the database driver and the database. Read "The Network," page 44.

For specific API code examples, read the chapter for the standards-based API that you work with:

- For ODBC users, see Chapter 5.
- For JDBC users, see Chapter 6.
- For ADO.NET users, see Chapter 7.

Understanding When the Driver Retrieves Data

You might think that if your application executes a query and then fetches one row of the results, the database driver only retrieves that one row. However, in most cases, that is not true; the driver retrieves many rows of data (a block of data) but returns only one row to the application. This is why the first fetch your

application performs may take longer than subsequent fetches. Subsequent fetches are faster because they do not require network round trips; the rows of data are already in memory on the client.

Some database drivers allow you to configure connection options that specify how much data to retrieve at a time. Retrieving more data at one time increases throughput by reducing the number of times the driver fetches data across the network when retrieving multiple rows. Retrieving less data at one time increases response time, because there is less of a delay waiting for the server to transmit data. For example, if your application normally fetches 200 rows, it is more efficient for the driver to fetch 200 rows at one time over the network than to fetch 50 rows at a time during four round trips over the network.

Retrieving Long Data

Retrieving long data—such as large XML data, long varchar/text, long varbinary, Clobs, and Blobs—across a network is slow and resource intensive. Do your application users really need to have the long data available to them? If yes, carefully think about the most optimal design. For example, consider the user interface of an employee directory application that allows the user to look up an employee's phone extension and department, and optionally, view an employee's photograph by clicking the name of the employee.

Employee	Phone	Dept
Harding	X4568	Manager
Hoover	X4324	Sales
Lincoln	X4329	Tech
Taft	X4569	Sales

Returning each employee's photograph would slow performance unnecessarily just to look up the phone extension. If users do want to see the photograph, they can click on the employee's name and the application can query the database again, specifying only the long columns in the Select list. This method allows users to return result sets without having to pay a high performance penalty for network traffic.

Having said this, many applications are designed to send a query such as `SELECT * FROM employees` and then request only the three columns they want

to see. In this case, the driver must still retrieve all the data across the network, including the employee photographs, even though the application never requests the photograph data.

Some database systems have optimized the expensive interaction between the database middleware and the database server when retrieving long data by providing an optimized database data type called LOBs (CLOB, BLOB, and so on). If your database system supports these data types and long data is created using those types, then the processing of queries such as `SELECT * FROM employees` is less expensive. Here's why. When a result row is retrieved, the driver retrieves only a placeholder for the long data (LOB) value. That placeholder is usually the size of an integer—very small. The actual long data (picture, document, scanned image, and so on) is retrieved only when the application specifically retrieves the value of the result column.

For example, if an employees table was created with the columns FirstName, LastName, EmpId, Picture, OfficeLocation, and PhoneNumber, and the Picture column is a long varbinary type, the following interaction would occur between the application, the driver, and the database server:

1. **Execute a statement**—The application sends a SQL statement (for example, `SELECT * FROM table WHERE ...`) to the database server via the driver.
2. **Fetch rows**—The driver retrieves all the values of all the result columns from the database server because the driver doesn't know which values the application will request. All values must be available when needed, which means that the entire image of the employee must be retrieved from the database server regardless of whether the application eventually processes it.
3. **Retrieve result values into the application**—When the application requests data, it is moved from the driver into the application buffers on a column-by-column basis. Even if result columns were prebound by the application, the application can still request result columns ad hoc.

Now suppose the employees table is created with the same columns except that the Picture field is a BLOB type. Now the following interaction would occur between the application, the driver, and the database server:

1. **Execute a statement**—The application sends a SQL statement (for example, `SELECT * FROM table WHERE ...`) to the database server via the driver.
2. **Fetch rows**—The driver retrieves all the values of all the result columns from the database server, as it did in the previous example. However, in this case, the entire employee image is not retrieved from the database server; instead, a placeholder integer value is retrieved.
3. **Retrieve result values into the application**—When the application requests data, it is moved from the driver into the application buffers on a column-by-column basis. If the application requests the contents of the Picture column, the driver initiates a request to the database server to retrieve the image of the employee that is identified by the placeholder value it retrieved. In this scenario, the performance hit associated with retrieving the image is deferred until the application actually requests that data.

In general, LOB data types are useful and preferred because they allow efficient use of long data on an as-needed basis. When the intent is to process large amounts of long data, using LOBs results in extra round trips between the driver and the database server. For example, in the previous example, the driver had to initiate an extra request to retrieve the LOB value when it was requested. These extra round trips usually are somewhat insignificant in the overall performance of the application because the number of overall round trips needed between the driver and the database server to return the entire contents of the long data is the expensive part of the execution.

Although you might prefer to use LOB types, doing so is not always possible because much of the data used in an enterprise today was not created yesterday. The majority of data you process was created long before LOB types existed, so the schema of the tables you use may not include LOB types even if they are supported by the version of the database system you are using. The coding techniques presented in this section are preferred regardless of the data types defined in the schema of your tables.

Performance Tip

Design your application to exclude long data from the Select list.

Limiting the Amount of Data Returned

One of the easiest ways to improve performance is to limit the amount of network traffic between the database driver and the database server—one way is to write SQL queries that instruct the driver to retrieve from the database and return to the application only the data that the application requires. However, some applications need to use SQL queries that generate a lot of traffic. For example, consider an application that needs to display information from support case histories, which each contain a 10MB log file. But, does the user really need to see the entire contents of the file? If not, performance would improve if the application displayed only the first 1MB of the log file.

Performance Tip

When you cannot avoid returning data that generates a lot of network traffic, control the amount of data being sent from the database to the driver by doing the following:

- Limiting the number of rows sent across the network
- Reducing the size of each row sent across the network

You can do this by using the methods or functions of the API you work with. For example, in JDBC, use `setMaxRows()` to limit the number of rows a query returns. In ODBC, call `SQLSetStmtAttr()` with the `SQL_ATTR_MAX_LENGTH` option to limit the number of bytes of data returned for a column value.

Choosing the Right Data Type

Advances in processor technology have brought significant improvements to the way that operations, such as floating-point math, are handled. However, when the active portion of your application does not fit into on-chip cache, retrieving and returning certain data types is expensive. 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. Table 2-1 lists the fastest to the slowest data types to process and explains why.

Table 2-1 Fastest to Slowest Processing of Data Types

Data Type	Processing
binary	Transfer of raw bytes from database to application buffers.
int, smallint, float	Transfer of fixed formats from database to application buffers.
decimal	Transfer of proprietary data from database to database driver. Driver must decode, which uses CPU, and then typically has to convert to a string. (Note: All Oracle numeric types are actually decimals.)
timestamp	Transfer of proprietary data from database to database driver. Driver must decode, which uses CPU, and then typically has to convert to a multipart structure or to a string. The difference between timestamp processing and decimal is that this conversion requires conversion into multiple parts (year, month, day, second, and so on).
char	Typically, transfer of larger amounts of data that must be converted from one code page to another, which is CPU intensive, not because of the difficulty, but because of the amount of data that must be converted.

Figure 2-8 shows a comparison of how many rows per second are returned when a column is defined as a 64-bit integer data type versus a decimal(20) data type. The same values are returned in each case. As you can see in this figure, many more rows per second are returned when the data is returned as an integer.

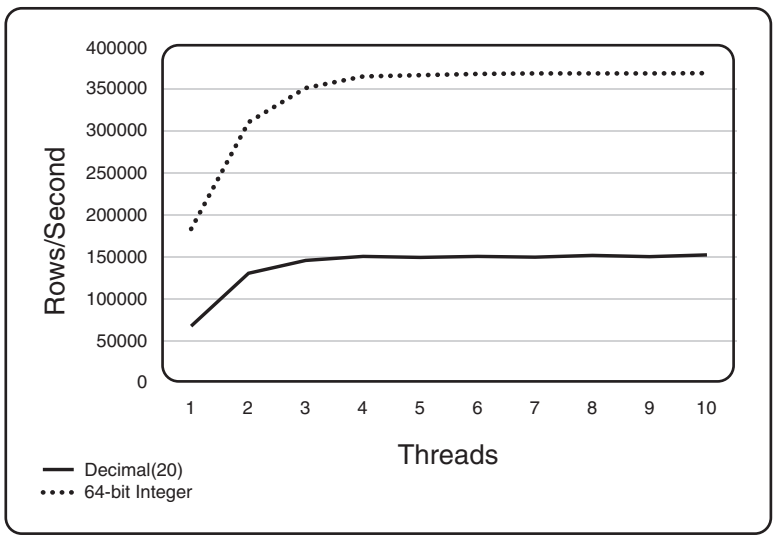

Figure 2-8 Comparison of different data types

Performance Tip

For multiuser, multivolume applications, it's possible that billions, or even trillions, of network packets move between the driver and the database server over the course of a day. Choosing data types that are processed efficiently can incrementally boost performance.

Using Scrollable Cursors

Scrollable cursors allow an application to go both forward and backward through a result set. However, because of limited support for server-side scrollable cursors in many database systems, drivers often emulate scrollable cursors, storing rows from a scrollable result set in a cache on the machine where the driver resides (client or application server). Table 2-2 lists five major database systems and explains their support of server-side scrollable cursors.

Table 2-2 Database Systems Support of Server-Side Scrollable Cursors

Database System	Explanation
Oracle	No native support of database server-side scrollable cursors. Drivers expose scrollable cursors to applications by emulating the functionality on the client.
MySQL	No native support of database server-side scrollable cursors. Drivers expose scrollable cursors to applications by emulating the functionality on the client.
Microsoft SQL Server	Server-side scrollable cursors are supported through stored procedures. Most drivers expose server-side cursors to applications.
DB2	Native support of some server-side scrollable cursor models. Some drivers support server-side scrollable cursors for the most recent DB2 versions. However, most drivers expose scrollable cursors to applications by emulating the functionality on the client.
Sybase ASE	Native support for server-side scrollable cursors was introduced in Sybase ASE 15. Versions prior to 15 do not natively support server-side scrollable cursors. Drivers expose scrollable cursors to applications by emulating the functionality on the client.

One application design flaw that we have seen many times is that an application uses a scrollable cursor to determine how many rows a result set contains even if server-side scrollable cursors are not supported in the database system. Here is an ODBC example; the same concept holds true for JDBC. Unless you are certain that the database natively supports using a scrollable result set, do not call `SQLExtendedFetch()` to find out how many rows the result set contains. For drivers that emulate scrollable cursors, calling `SQLExtendedFetch()` results in the driver returning all results across the network to reach the last row.

This emulated model of scrollable cursors provides flexibility for the developer but comes with a performance penalty until the client cache of rows is fully populated. Instead of using a scrollable cursor to determine the number of rows, count the rows by iterating through the result set or get the number of rows by submitting a `Select` statement with the `Count` function. For example:

```
SELECT COUNT(*) FROM employees WHERE ...
```

Extended Security

It is no secret that performance penalties are a side effect of extended security. If you've ever developed an application that required security, we're sure that you've discovered this hard truth. We include this section in the book simply to point out the penalties that go along with security and to provide suggestions for limiting these penalties if possible.

In this section, we discuss two types of security: network authentication and data encryption across the network (as opposed to data encrypted in the database).

If your database driver of choice does not support network authentication or data encryption, you cannot use this functionality in your database application.

Network Authentication

On most computer systems, an encrypted password is used to prove a user's identity. If the system is a distributed network system, this password is transmitted over the network and can possibly be intercepted and decrypted by malicious hackers. Because this password is the one secret piece of information that identifies a user, anyone knowing a user's password can effectively *be* that user.

In your enterprise, the use of passwords may not be secure enough. You might need network authentication.

Kerberos, a network authentication protocol, provides a way to identify users. Any time users request a network service, such as a database connection, they must prove their identity.

Kerberos was originally developed at MIT as a solution to the security issues of open network computing environments. Kerberos is a trusted third-party authentication service that verifies users' identities.

Kerberos keeps a database (the Kerberos server) of its clients and their private keys. The **private key** is a complex formula-driven value known only to Kerberos and the client to which it belongs. If the client is a user, the private key is an encrypted password.

Both network services that require authentication and clients who want to use these services must register with Kerberos. Because Kerberos knows the private keys of all clients, it creates messages that validate the client to the server and vice versa.

In a nutshell, here is how Kerberos works:

1. **The user obtains credentials that are used to request access to network services**. These credentials are obtained from the Kerberos server and are in the form of a Ticket-Granting Ticket (TGT). This TGT authorizes the Kerberos server to grant the user a service ticket, which authorizes his access to network services.
2. **The user requests authentication for a specific network service**. The Kerberos server verifies the user's credentials and sends a service ticket to him.
3. **The user presents the service ticket to the end server.** If the end server validates the user, the service is granted.

Figure 2-9 shows an example of requesting a database connection (a network service) when using Kerberos.

An application user requests a database connection after a TGT has been obtained:

1. The application sends a request for a database connection to the Kerberos server.
2. The Kerberos server sends back a service ticket.
3. The application sends the service ticket to the database server.
4. The database server validates the client and grants the connection.

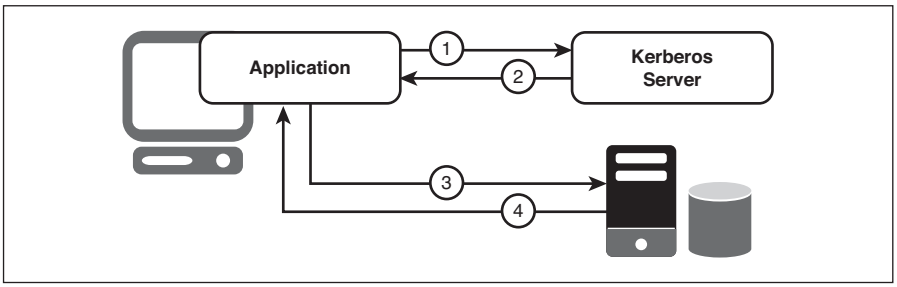

Figure 2-9 Kerberos

Even when you don't use Kerberos, database connections are performance-expensive; they can require seven to ten network round trips (see the section, "Why Connections Are Performance-Expensive," page 11, for more details). Using Kerberos comes with the price of adding more network round trips to establish a database connection.

Performance Tip

To get the best performance possible when using Kerberos, place the Kerberos server on a dedicated machine, reduce the networking services run on this machine to the absolute minimum, and make sure you have a fast, reliable network connection to the machine.

Data Encryption Across the Network

If your database connection is not configured to use data encryption, data is sent across the network in a "native" format; for example, a 4-byte integer is sent across the network as a 4-byte integer. The native format is defined by either of the following:

- The database vendor
- The database driver vendor in the case of a driver with an independent protocol architecture such as a Type 3 JDBC driver

The native format is designed for fast transmission and can be decoded by interceptors given some time and effort.

Because a native format does not provide complete protection from interceptors, you may want to use data encryption to provide a more secure transmission of data. For example, you may want to use data encryption in the following scenarios:

- You have offices that share confidential information over an intranet.
- You send sensitive data, such as credit card numbers, over a database connection.
- You need to comply with government or industry privacy and security requirements.

Data encryption is achieved by using a protocol for managing the security of message transmission, such as Secure Sockets Layer (SSL). Some database systems, such as DB2 for z/OS, implement their own data encryption protocol. The way the database-specific protocols work and the performance penalties associated with them are similar to SSL.

In the world of database applications, **SSL** is an industry-standard protocol for sending encrypted data over database connections. SSL secures the integrity of your data by encrypting information and providing client/server authentication.

From a performance perspective, SSL introduces an additional processing layer, as shown in Figure 2-10.

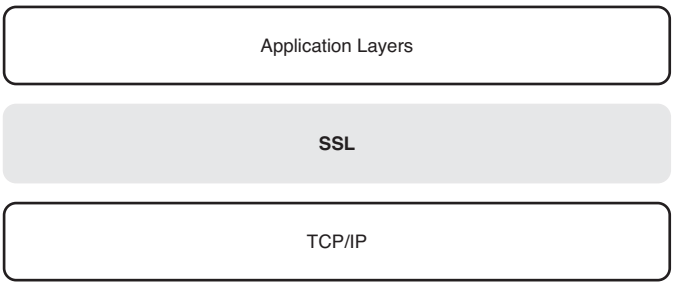

Figure 2-10 SSL: an additional processing layer

The SSL layer includes two CPU-intensive phases: SSL handshake and encryption.

When encrypting data using SSL, the database connection process includes extra steps between the database driver and the database to negotiate and agree

upon the encryption/decryption information that will be used. This is called the SSL handshake. An SSL handshake results in multiple network round trips as well as additional CPU to process the information needed for every SSL connection made to the database.

During an SSL handshake, the following steps take place, as shown in Figure 2-11:

1. The application via a database driver sends a connection request to the database server.
2. The database server returns its certificate and a list of supported encryption methods (cipher suites).
3. A secure, encrypted session is established when both the database driver and the server have agreed on an encryption method.

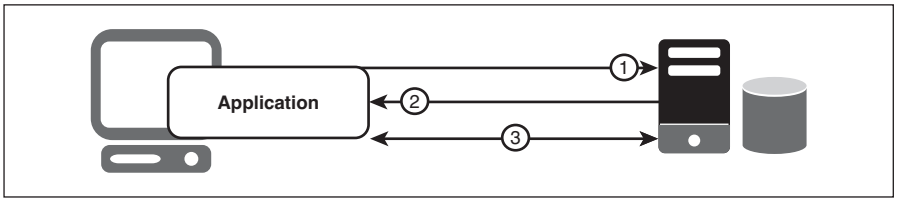

Figure 2-11 SSL handshake

Encryption is performed on each byte of data transferred; therefore, the more data being encrypted, the more processing cycles occur, which means slower network throughput.

SSL supports symmetric encryption methods such as DES, RC2, and Triple DES. Some of these symmetric methods cause a larger performance penalty than others, for example, Triple DES is slower than DES because larger keys must be used to encrypt/decrypt the data. Larger keys mean more memory must be referenced, copied, and processed. You cannot always control which encryption method your database server uses, but it is good to know which one is used so that you can set realistic performance goals.

Figure 2-12 shows an example of how an SSL connection can affect throughput. In this example, the same benchmark was run twice using the same application, JDBC driver, database server, hardware, and operating system. The only variable was whether an SSL connection was used.

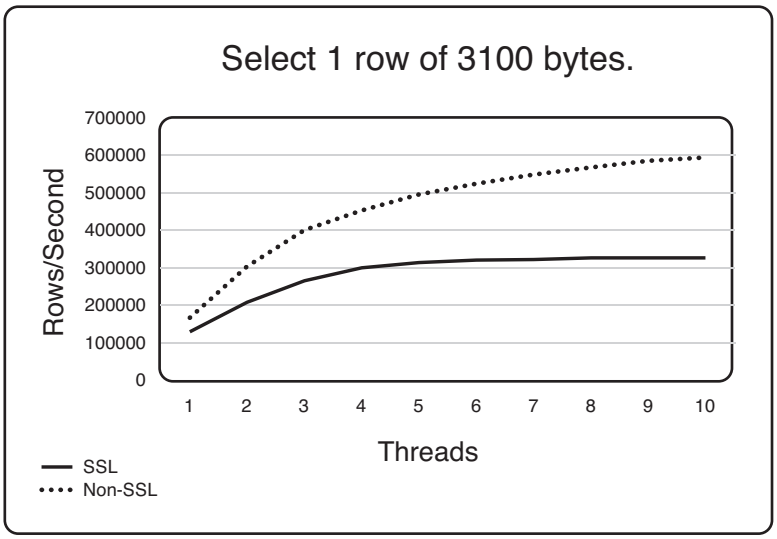

Figure 2-12 Rows per second: SSL versus non-SSL

Figure 2-13 shows the CPU associated with the throughput of this example. As you can see, CPU use increases when using an SSL connection.

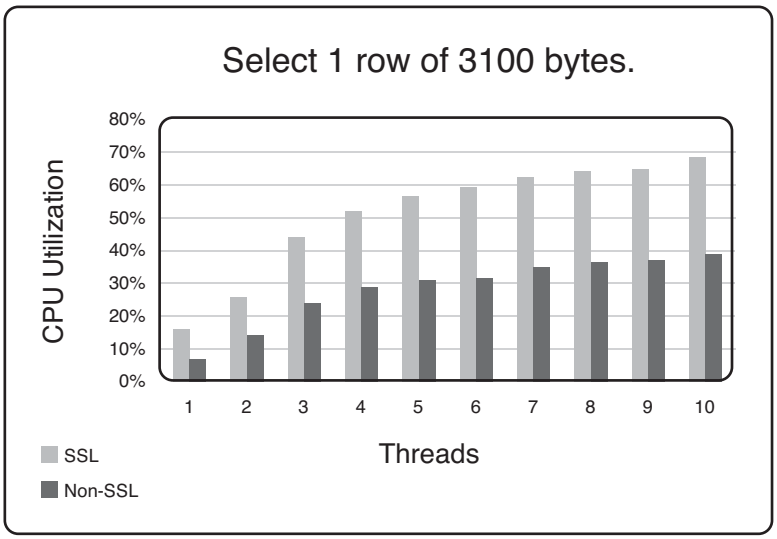

Figure 2-13 CPU utilization: SSL versus non-SSL

Performance Tip

To limit the performance penalty associated with data encryption, consider establishing a connection that uses encryption for accessing sensitive data such as an individual's tax ID number, and another connection that does not use encryption for accessing data that is less sensitive, such as an individual's department and title. There is one caveat here: Not all database systems allow this. Oracle and Microsoft SQL Server are examples of database systems that do. Sybase is an example of either all connections to the database use encryption or none of them do.

Static SQL Versus Dynamic SQL

At the inception of relational database systems and into the 1980s, the only portable interface for applications was embedded SQL. At that time, there was no common function API such as a standards-based database API, for example, ODBC. **Embedded SQL** is SQL statements written within an application programming language such as C. These statements are preprocessed by a SQL preprocessor, which is database dependent, before the application is compiled. In the preprocessing stage, the database creates the access plan for each SQL statement. During this time, the SQL was embedded and, typically, always static.

In the 1990s, the first portable database API for SQL was defined by the SQL Access Group. Following this specification came the ODBC specification from Microsoft. The ODBC specification was widely adopted, and it quickly became the de facto standard for SQL APIs. Using ODBC, SQL did not have to be embedded into the application programming language, and precompilation was no longer required, which allowed database independence. Using SQL APIs, the SQL is not embedded; it is dynamic.

What is static SQL and dynamic SQL? **Static SQL** is SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application. **Dynamic SQL** is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries. Thus, the SQL statements cannot be hard-coded into the application.

Static SQL provides performance advantages over dynamic SQL because static SQL is preprocessed, which means the statements are parsed, validated, and optimized only once.

If you are using a standards-based API, such as ODBC, to develop your application, static SQL is probably not an option for you. However, you can achieve a similar level of performance by using either statement pooling or stored procedures. See "Statement Pooling," page 29, for a discussion about how statement pooling can improve performance.

A **stored procedure** is a set of SQL statements (a subroutine) available to applications accessing a relational database system. Stored procedures are physically stored in the database. The SQL statements you define in a stored procedure are parsed, validated, and optimized only once, as with static SQL.

Stored procedures are database dependent because each relational database system implements stored procedures in a proprietary way. Therefore, if you want your application to be database independent, think twice before using stored procedures.

Note

Today, a few tools are appearing on the market that convert dynamic SQL in a standards-based database application into static SQL. Using static SQL, applications achieve better performance and decreased CPU costs. The CPU normally used to prepare a dynamic SQL statement is eliminated.

The Network

The network, which is a component of the database middleware, has many factors that affect performance: database protocol packets, network packets, network hops, network contention, and packet fragmentation. See "Network," in Chapter 4 (page 86) for details on how to understand the performance implications of the network and guidelines for dealing with them.

In this section, let's look at one important fact about performance and the network: database application performance improves when communication between the database driver and the database is optimized.

With this in mind, you should always ask yourself: How can I reduce the information that is communicated between the driver and the database? One important factor in 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. If the database server accepts a 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.

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—increasing the number of packets sent over the network to return the same amount of data to the client (as shown in Figure 2-14).

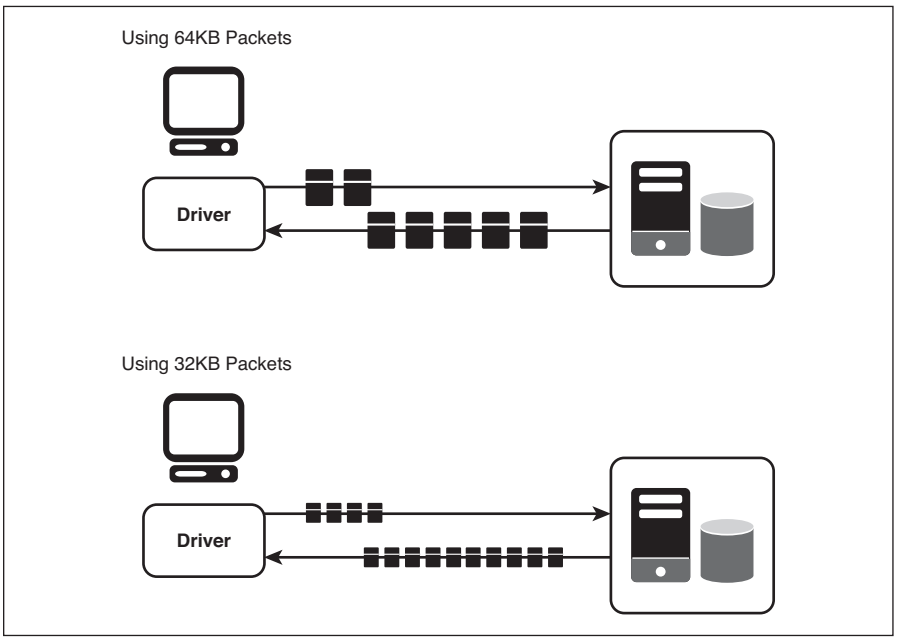

Figure 2-14 Using different packet sizes

This 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.

You might be thinking, "But how can I do anything about the size of database protocol packets?" You can use a database driver that allows you to configure their size. See "Runtime Performance Tuning Options," page 62, for more information about which performance tuning options to look for in a database driver.

The Database Driver

The database driver, which is a component of the database middleware, can degrade the performance of your database application because of the following reasons:

- The architecture of the driver is not optimal.
- The driver is not tunable. It does not have runtime performance tuning options that allow you to configure the driver for optimal performance.

See Chapter 3, "Database Middleware: Why It's Important," for a detailed description of how a database driver can improve the performance of your database application.

In this section, let's look at one important fact about performance and a database driver: The architecture of your database driver matters. Typically, the most optimal architecture is database wire protocol.

Database wire protocol drivers communicate with the database directly, eliminating the need for the database's client software, as shown in Figure 2-15.

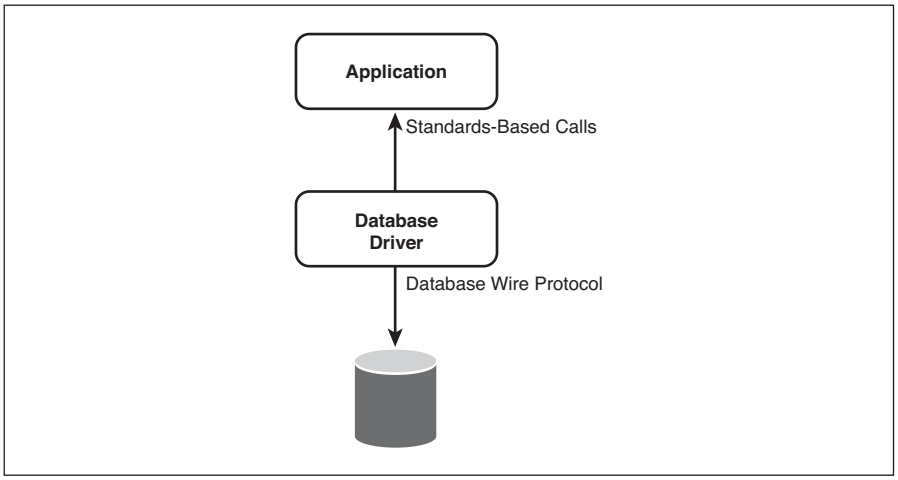

Figure 2-15 Database wire protocol architecture

Using a wire protocol database driver improves the performance of your database application because it does the following:

- Decreases latency by eliminating the processing required in the client software and the extra network traffic caused by the client software.
- Reduces network bandwidth requirements from extra transmissions. That is, database wire protocol drivers optimize network traffic because they can control interaction with TCP.

We go into more detail about the benefits of using a database wire protocol driver in "Database Driver Architecture," page 55.

Know Your Database System

You may think your database system supports all the functionality that is specified in the standards-based APIs (such as ODBC, JDBC, and ADO.NET). That is likely not true. Yet, the driver you use may provide the functionality, which is often a benefit to you. For example, if your application performs bulk inserts or updates, you can improve performance by using arrays of parameters. Yet, not all database systems support arrays of parameters. In any case, if you use a database driver that supports them, you can use this functionality even if the database system does not support it, which 1) results in performance improvements for bulk inserts or updates, and 2) eliminates the need for you to implement the functionality yourself.

The trade-off of using functionality that is not natively supported by your database system is that emulated functionality can increase CPU use. You must weigh this trade-off against the benefit of having the functionality in your application.

The protocol of your database system is another important implementation detail that you should understand. Throughout this chapter, we discussed design decisions that are affected by the protocol used by your database system of choice: cursor-based or streaming. Explanations of these two protocols can be found in "One Connection for Multiple Statements" on page 16.

Table 2-3 lists some common functionality and whether it is natively supported by five major database systems.

Table 2-3 Database System Native Support

Functionality	DB2	Microsoft SQL Server	MySQL	Oracle	Sybase ASE
Cursor-based protocol	Supported	Supported	Not supported	Supported	Not supported
Streaming protocol	Not supported	Not supported	Supported	Not supported	Supported
Prepared statements	Native	Native	Native	Native	Not supported
Arrays of parameters	Depends on version	Depends on version	Not supported	Native	Not supported
Scrollable cursors[1]	Supported	Supported	Not supported	Not supported	Depends on version
Auto-commit mode	Not supported	Not supported	Native	Native	Native
LOB locators	Native	Native	Not supported	Native	Not supported

[1] See Table 2-2, page 36, for more information about how these database systems support scrollable cursors.

Using Object-Relational Mapping Tools

Most business applications access data in relational databases. However, the relational model is designed for efficiently storing and retrieving data, not for the object-oriented model often used for business applications.

As a result, new object-relational mapping (ORM) tools are becoming popular with many business application developers. Hibernate and Java Persistence API (JPA) are such tools for the Java environment, and NHibernate and ADO.NET Entity Framework are such tools for the .NET environment.

Object-relational mapping tools map object-oriented programming objects to the tables of relational databases. When using relational databases with objects, typically, an ORM tool can reduce development costs because the tool does the object-to-table and table-to-object conversions needed. Otherwise, these conversions must be written in addition to the application development. ORM tools allow developers to focus on the business application.

From a design point of view, you need to know that when you use object-relational mapping tools you lose much of the ability to tune your database application code. For example, you are not writing the SQL statements that are sent to the database; the ORM tool is creating them. This can mean that the SQL statements could be more complex than ones you would write, which can result in performance issues. Also, you don't get to choose the API calls used to return data, for example, SQLGetData versus SQLBindCol for ODBC.

To optimize application performance when using an ORM tool, we recommend that you tune your database driver appropriately for use with the database your application is accessing. For example, you can use a tool to log the packets sent between the driver and the database and configure the driver to send a packet size that is equal to the packet size of that configured on the database. See Chapter 4, "The Environment: Tuning for Performance," for more information.

Summary

Many factors affect performance. Some are beyond your control, but thoughtful design of your application and the configuration of the database middleware that connects your application to the database server can result in optimal performance.

If you are going to design only one aspect of your application, let it be database connections, which are performance-expensive. Establishing a connection can take up to ten network round trips. You should assess whether connection pooling or one connection at a time is more appropriate for your situation.

When designing your database application, here are some important questions to ask: Are you retrieving only the minimum amount of data that you need? Are you retrieving the most efficient data type? Would a prepared statement save you some overhead? Could you use a local transaction instead of a more performance-expensive distributed transaction?

Lastly, make sure that you are using the best database driver for your application. Does your database driver support all the functionality that you want to use in your application? For example, does your driver support statement pooling? Does the driver have runtime performance tuning options that you can configure to improve performance? For example, can you configure the driver to reduce network activity?