5. Application Design

Application design is more than just writing efficient database requests in application programs. Every aspect of the way the program is coded affects the usability and effectiveness of the application. Of course, application design includes database concerns such as interfacing SQL with traditional programming languages and the type of SQL to use. However, each application program must be designed to ensure the integrity of the data it modifies. Additionally, performance has to be treated as a design issue.

Performance has to be treated as a design issue.

At the forefront the DBA must promote the concept of application design based on thorough knowledge of the database. It is unacceptable to allow programmers to design and code applications without considering how the programs will perform as they interact with databases. Some uninformed organizations approach database application development with no proactive performance engineering: The assumption is that any performance problems can be resolved after development by the DBA. However, it may be impossible to tune an improperly designed application program without rewriting it. So why not write it correctly the first time?

The intent of this chapter is not to teach software development methodology or to provide an in-depth treatise on programming—nor is it a primer on SQL. The focus of discussion will be high-level application design issues that need to be understood when writing applications that use a database for persistent storage of data. All DBAs should understand the concepts in this chapter and be able to effectively communicate them to the developers in their organization.

Chapter 12, "Application Performance," provides additional coverage of application performance issues as they pertain to database development.

Database Application Development and SQL

Designing a proper database application system is a complex and time-consuming task. The choices made during application design will impact the usefulness of the final, delivered application. Indeed, an improperly designed and coded application may need to be redesigned and recoded from scratch if it is inefficient, ineffective, or not easy to use.

Designing a proper database application system is a complex task.

To properly design an application that relies on databases for persistent data storage,

the system designer at a minimum will need to understand the following issues:

- How data is stored in a relational database
- How to code SQL statements to access and modify data in the database
- How SQL differs from traditional programming languages
- How to embed SQL statements into a host programming language
- How to optimize database access by changing SQL and indexes
- Programming methods to avoid potential database processing problems

In general, the developer must match the application development languages and tools to the physical database design and functionality of the DBMS. The first task is to master the intricacies of SQL.

SQL

Structured Query Language, better known as SQL (and pronounced "sequel" or "ess-cue-el"), is the de facto standard for accessing relational databases. All RDBMS products, and even some nonrelational DBMS products, use SQL to manipulate data. SQL is the de facto standard for accessing relational databases.

Why is SQL so pervasive within the realm of relational data access? There are many reasons for SQL's success. Foremost is that SQL is a high-level language that provides a greater degree of abstraction than traditional procedural languages. Third-generation languages, such as COBOL and C, and even fourth-generation languages, usually require the programmer to navigate data structures. Program logic must be coded to proceed record by record through data stores in an order determined by the application programmer or systems analyst. This information is encoded in the high-level language and is difficult to change after it has been programmed.

SQL, by contrast, is designed such that programmers specify *what* data is needed. It does not—indeed it cannot—specify *how* to retrieve it. SQL is coded without embedded data-navigational instructions. The DBMS analyzes each SQL statement and formulates data-navigational instructions "behind the scenes." These data-navigational instructions are commonly called *access paths*. A heavy burden is removed from the programmer by empowering the DBMS to determine the optimal access paths to the data. Because the DBMS better understands the state of the data it stores, it can produce a more efficient and dynamic access path to the data. The result is that SQL, used properly, provides a quicker application development and

prototyping environment than is available with corresponding high-level languages. Furthermore, as the data characteristics and access patterns change, the DBMS can change access paths for SQL queries without requiring the actual SQL to be changed in any way.

SQL specifies what data is needed . . . not how to retrieve it.

Inarguably, though, the single most important feature that has solidified SQL's success is its capability to retrieve data easily using English-like syntax. It is much easier to understand a query such as

```
SELECT    deptnum, deptname
FROM      dept
WHERE     supervisornum = '903';
```

than it is to understand pages and pages of C or BASIC source code, let alone the archaic instructions of Assembler. Because SQL programming instructions are easier to understand, they are easier to learn and maintain—affording users and programmers more productivity in a shorter period of time. However, do not underestimate SQL: Mastering all of its intricacies is not easy and will require much study and practice.

SQL also uses a free-form structure that makes it very flexible. The SQL programmer has the ability to develop SQL statements in a way best suited to the given user. Each SQL request is parsed by the DBMS before execution to check for proper syntax and to optimize the request. Therefore, SQL statements do not need to start in any given column and can be strung together on one line or broken apart on several lines. For example, the following SQL statement:

Click here to view code image

```
SELECT deptnum, deptname FROM dept WHERE supervisornum = '903';
```

is exactly equivalent to the previous SQL statement shown. Another example of SQL's flexibility is that the programmer can formulate a single request in a number of different and functionally equivalent ways—a feature that also can be very confusing for SQL novices. Furthermore, the flexibility of SQL is not always desirable, because different but logically equivalent SQL formulations can result in differing performance results. Refer to the sidebar "Joins versus Subqueries" for an example.

Finally, one of the greatest benefits derived from using SQL is its ability to operate on sets of data with a single line of code. Multiple rows can be retrieved, modified, or removed in one fell swoop by using a single SQL statement. This feature provides the

SQL developer with great power but also limits the overall functionality of SQL. Without the ability to loop or step through multiple rows one at a time, certain tasks are impossible to accomplish using only SQL. Of course, as more and more functionality is added to SQL, the number of tasks that can be coded using SQL alone is increasing. For example, SQL can be used to create a stored procedure to perform many programming tasks that formerly required a traditional programming language to accomplish. Furthermore, most of the popular relational DBMS products support extended versions of SQL with procedural capabilities. Table 5.1 details the most popular procedural SQL dialects.

Table 5.1. SQL Usage Considerations

DBMS	Procedural SQL Dialect
Oracle	PL/SQL
Microsoft SQL Server	Transact-SQL
Sybase Adaptive Server Enterprise	Transact-SQL
DB2	SQL Procedure Language

Set-at-a-Time Processing and Relational Closure

Every operation performed on a relational database operates on a table (or set of tables) and results in another table. This feature of relational databases is called *relational closure*. All SQL data manipulation operations—that is, SELECT, INSERT, UPDATE, and DELETE statements—are performed at a set level. One retrieval statement can return multiple rows; one modification statement can modify multiple rows.

All SQL data manipulation operations are performed at a set level.

To clarify the concept of relational closure, refer to Figure 5.1. A database user initiates SQL requests. Each SQL statement can access one or many tables in the database. The SQL statement is sent to the DBMS, whereupon the query is analyzed, optimized, and executed. The DBMS formulates the access path to the data, and upon completion of the request the desired information is presented to the user as a set of columns and rows—in other words, a table. The result will consist of one or more columns with zero, one, or many rows. Because SQL performs set-level processing, the DBMS operates on a set of data, and a set of data is always returned as the result. Of course, the results set can be empty, or it can contain only one row or column. The relational model and set-level processing are based on the mathematical laws of *set theory*, which permit empty and single-valued sets.

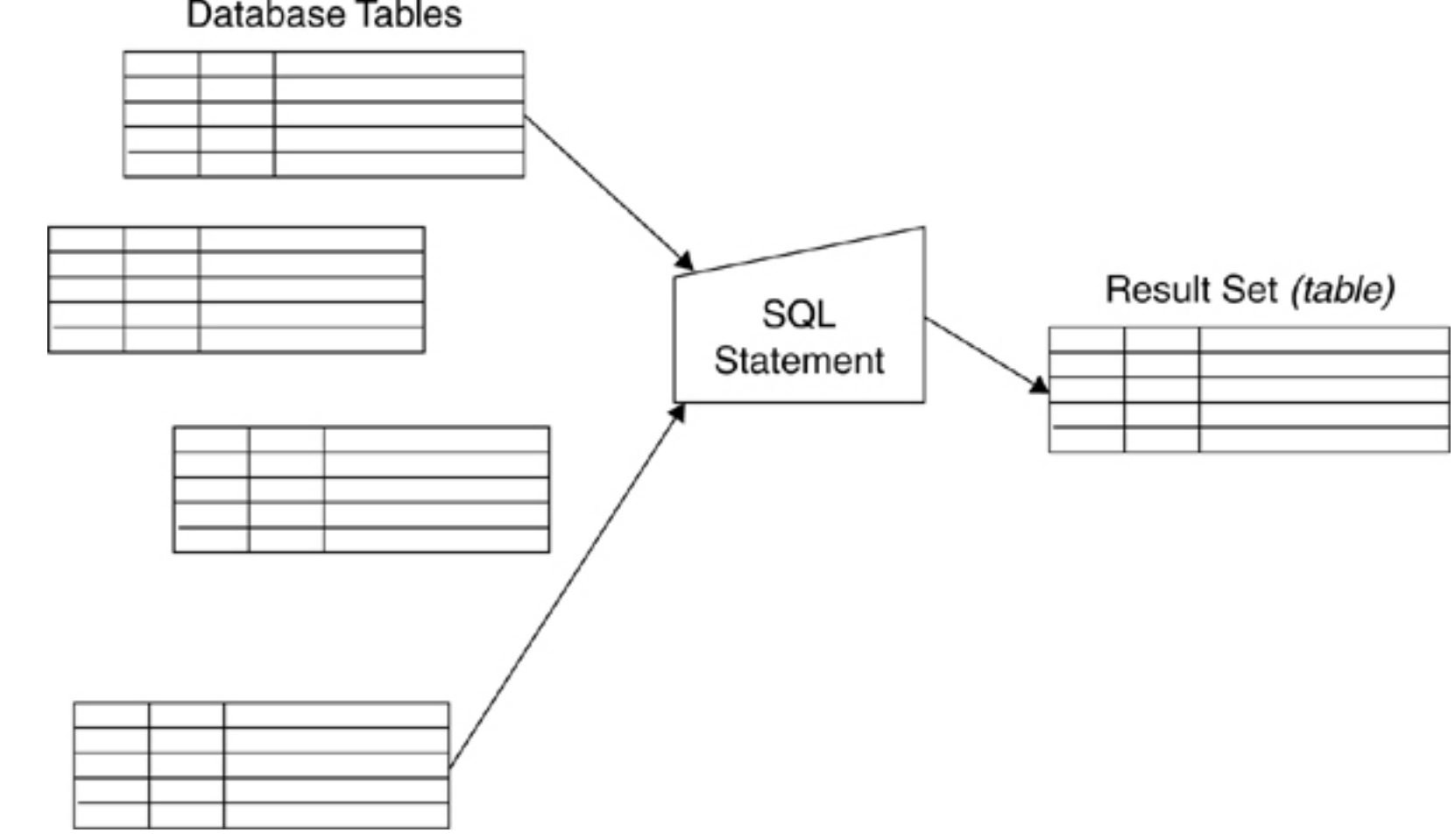

Figure 5.1. Relational closure

Application developers face a potential problem when using relational databases because of the set-at-a-time nature of SQL. Most programming languages operate on data one record at a time. When a program requires relational data, though, it must request the data using SQL. This creates an impedance mismatch. The program expects data to be returned a single row at a time, but SQL returns data a set at a time. There are different ways to get around this mismatch, depending on the DBMS, programming language, and environment. Most DBMS products provide a feature called a *cursor* that accepts the input from a SQL request and provides a mechanism to fetch individual rows of the results set. Some programming environments and fourth-generation languages automatically transform multirow sets to single rows when communicating with the DBMS.

Furthermore, most programmers are accustomed to hard-wiring data-navigational instructions into their programs. SQL specifies what to retrieve but not how to retrieve it. The DBMS determines how best to retrieve the data. Programmers unaccustomed to database processing are unlikely to grasp this concept without some training. At any rate, programmers will need to be trained in these high-level differences between non-database and database programming techniques. This job usually falls upon the DBA or other highly skilled database technician within the organization. Of course, there are many more details at a lower level that the database programmer needs to know, such as SQL syntax, debugging and testing methods, optimization techniques, and program preparation procedures (compilation, bind, etc.).

The DBMS determines how best to retrieve the data.

Embedding SQL in a Program

Most database applications require a host programming language to use SQL to communicate with the database. A wide range of programming languages can be used with SQL, from traditional languages such as COBOL, FORTRAN, and Assembler to more modern languages such as C/C++, Java, PHP, and Visual Basic. Your choice of host programming language can impact the way you will have to code SQL. For example, SQL is embedded directly into a COBOL program, whereas a language like C requires an API such as ODBC to issue SQL statements.

The choice of development language will likely be limited to just a few at your shop. You should attempt to minimize the number of different languages you use, because it will make supporting and maintaining your applications easier. Furthermore, such limitations will make it easier for DBAs to administer and optimize the database environment. A DBA should be capable of reading and understanding program code for each language used to access databases within the organization.

Minimize the number of different languages you use.

Some application development projects use an *IDE* (*integrated development environment*) or *code generator* to create programs from program specifications. Exercise caution when using this approach: Don't allow the code generator to create SQL for you without first testing it for efficiency. Poor performance can result when using a code generation tool because these tools often have very little knowledge of the DBMS you are using. In most cases the code generator is designed to work for multiple DBMS products and is therefore not optimized for any of them. Test the SQL that is generated and, if necessary, modify the generated SQL or build indexes to optimize the generated SQL before moving the programs to a production environment. The DBA should be responsible for implementing a procedure to ensure that such SQL performance testing occurs.

SQL Middleware and APIs

Application programs require an *interface* for issuing SQL to access or modify data. The interface is used to embed SQL statements in a host programming language, such as COBOL, Java, C, or Visual Basic. Standard interfaces enable application programs to access databases using SQL. There are several popular standard interfaces or APIs for database programming, including ODBC, JDBC, SQLJ, and OLE DB.

One of the most popular SQL APIs is Open Database Connectivity (ODBC). Instead of directly embedding SQL in the program, ODBC uses *callable routines*. ODBC provides routines to allocate and deallocate resources, control connections to the database, execute SQL statements, obtain diagnostic information, control transaction

termination, and obtain information about the implementation. ODBC is basically a call-level interface (CLI) for interacting with databases. The CLI issues SQL statements against the database by using procedure calls instead of direct embedded SQL statements.

ODBC is basically a call-level interface for interacting with databases.

Microsoft invented the ODBC interface to enable relational database access for Microsoft Windows programming. However, ODBC has become an industry-standard CLI for SQL programming. Indeed, every major DBMS today supports ODBC.

ODBC relies on *drivers*, which are optimized ODBC interfaces for a particular DBMS implementation. Programs can make use of the ODBC drivers to communicate with any ODBC-compliant database. The ODBC drivers enable a standard set of SQL statements in any Windows application to be translated into commands recognized by a remote SQL-compliant database.

Another popular SQL API is Java Database Connectivity (JDBC). JDBC enables Java to access relational databases. Similar to ODBC, JDBC consists of a set of classes and interfaces that can be used to access relational data. There are several types of JDBC middleware, including the JDBC-to-ODBC bridge, as well as direct JDBC connectivity to the relational database. Anyone familiar with application programming and ODBC (or any call-level interface) can get up and running with JDBC quickly. (Refer to Chapter 21, "Database Connectivity," for a discussion of the various types of JDBC drivers.)

Another way to access databases from a Java program is by using SQLJ. SQLJ enables developers to directly embed SQL statements in Java programs, thus providing static SQL support to Java. A precompiler translates the embedded SQL into Java code. The Java program is then compiled into bytecodes, and a database bind operation creates packaged access routines for the SQL.

SQLJ enables developers to embed SQL statements in Java programs.

OLE DB, which stands for Object Linking and Embedding Database, is an interface that is based on the COM architecture. It is a low-level interface to data. OLE DB provides applications with uniform access to data stored in diverse information sources. It allows greater flexibility than ODBC because it can be used to access both relational and nonrelational data. OLE DB presents an object-oriented interface for generic data access. OLE DB is conceptually divided into *consumers* and *providers*. The consumers are the applications that need access to the data, and the providers are the software components that implement the interface and thereby provide the data to the consumer.

SQL Server 2012[1] is planned to be the final version to include an OLE DB provider for SQL Server. However, Microsoft has indicated that support will continue for at least seven years.

COM is Microsoft's component-based development architecture. Using COM, developers can create application components that can be pieced together to create application systems. The components can be written by different developers and need not be written using the same programming language. ADO (which stands for ActiveX Data Objects) is a set of software components that programmers can use to access data and data services.

Both COM and ADO predate the .NET framework but have been adapted for use by .NET.

Application Infrastructure

Application infrastructure is the combined hardware and software environment that supports and enables the application. The application infrastructure will vary from organization to organization, and even from application to application within an organization. The application infrastructure provides the foundation for building, deploying, and managing applications with high performance, security, and control.

From a hardware perspective, the application infrastructure includes the servers, clients, and networking components. From a software perspective, things are a bit more difficult to nail down. Software components of an application infrastructure can include database servers, application servers, Web servers, transaction managers, and development frameworks. Some of the key functionality of application infrastructure includes transaction management, clustering, reliable application-to-application messaging, system management, advanced application development tools, proprietary access, and interoperability with legacy technologies.

From a mainframe perspective, the application infrastructure may consist of IBM z Series hardware running z/OS, DB2, CICS, with application programs written in COBOL. Typically, applications consist of both batch and online workload. A modern mainframe infrastructure adds interfaces to non-mainframe clients, as well as WebSphere Application Server and Java programs. Most new mainframe development uses IDEs to code modern applications instead of relying upon COBOL programmers.

Most modern, distributed, non-mainframe application development projects typically rely upon application development frameworks. The two most commonly used frameworks are Microsoft .NET and J2EE.

.NET

The Microsoft .NET framework provides a comprehensive development platform for the construction, deployment, and management of applications. The .NET framework provides CLR (common language run time) and a class library for building components using a common foundation. This offers benefits to developers such as support for standard practices, extensibility, and a tightly integrated set of development tools.

The .NET framework consists of multiple major components in addition to the CLR and class library. From a data perspective, the most important component is ADO.NET, which provides access to data sources such as a database management system.

See Figure 5.2 for a depiction of the .NET framework and its components.

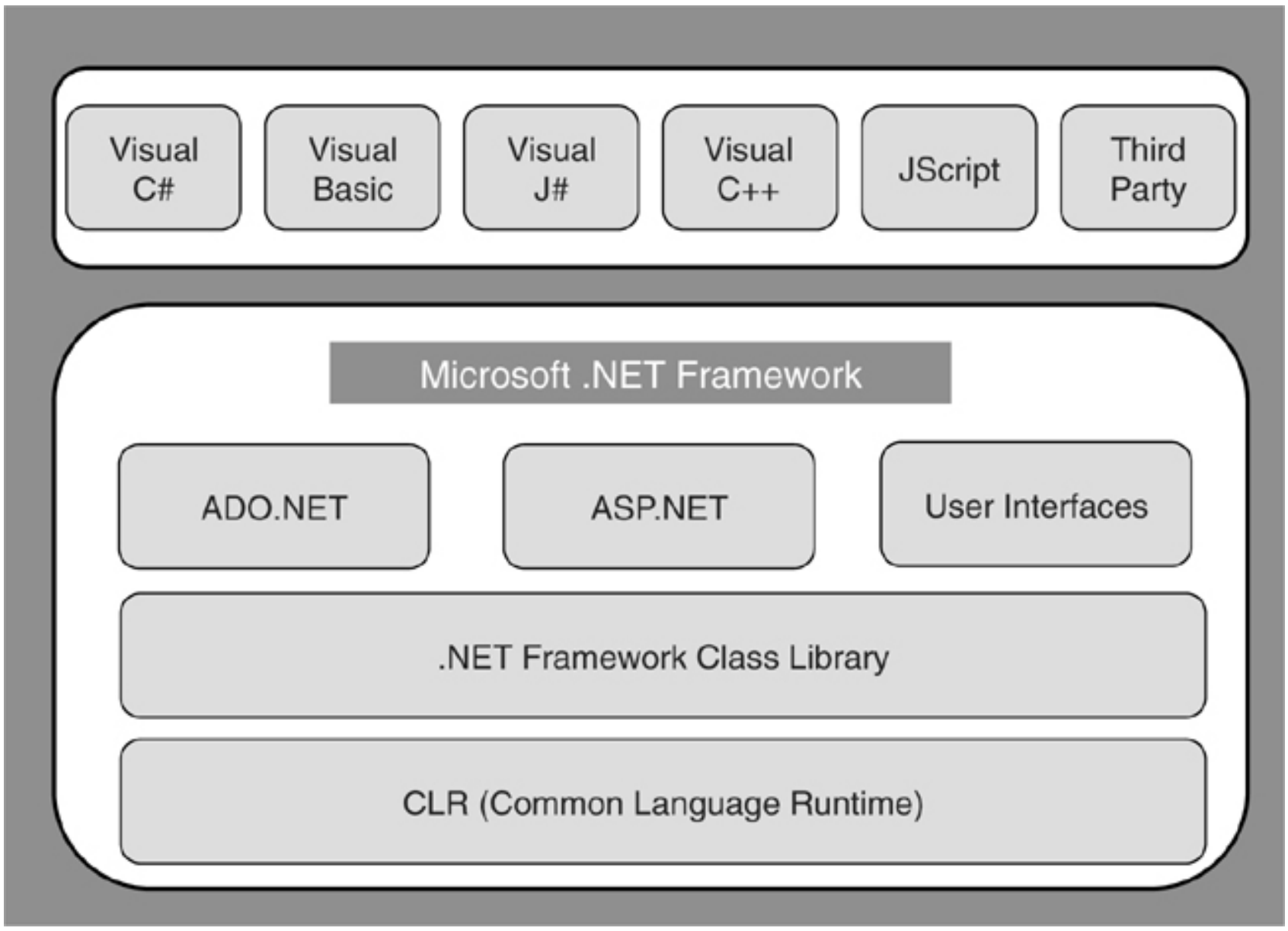

Figure 5.2. The .NET framework

ADO.NET is composed of a series of technologies that enable .NET developers to interact with data in standard, structured, and predominantly disconnected ways. Applications that use ADO.NET depend on .NET class libraries provided in DLL files. ADO.NET manages both internal data (created in memory and used by the program) and external data (in the database). It provides interoperability and maintainability through its use and support of XML, simplified programmability with a programming model that uses strongly typed data, and enhanced performance and scalability.

J2EE and Java

The Java 2 Platform, Enterprise Edition (J2EE), is a set of coordinated specifications and practices that together enable solutions for developing, deploying, and managing multitier enterprise applications. The J2EE platform simplifies enterprise applications by basing them on standardized, modular components. J2EE provides a complete set of services to those components and handles many details of application construction without requiring complex programming.

So J2EE is not exactly a software framework, but a set of specifications, each of which dictates how various J2EE functions must operate. Software conforming to the J2EE platform offers advantages such as "Write Once, Run Anywhere" portability, JDBC API for database access, CORBA technology for interaction with existing enterprise resources, and a security model for data protection. Building on this base, the Java 2 Platform, Enterprise Edition, adds full support for Enterprise JavaBeans components, Java Servlets API, JavaServer Pages, and XML technology.

See Figure 5.3 for a depiction of a sample J2EE implementation. Additional information and clarification on J2EE can be found online at http://java.sun.com/j2ee/reference/whitepapers/j2ee_guide.pdf.

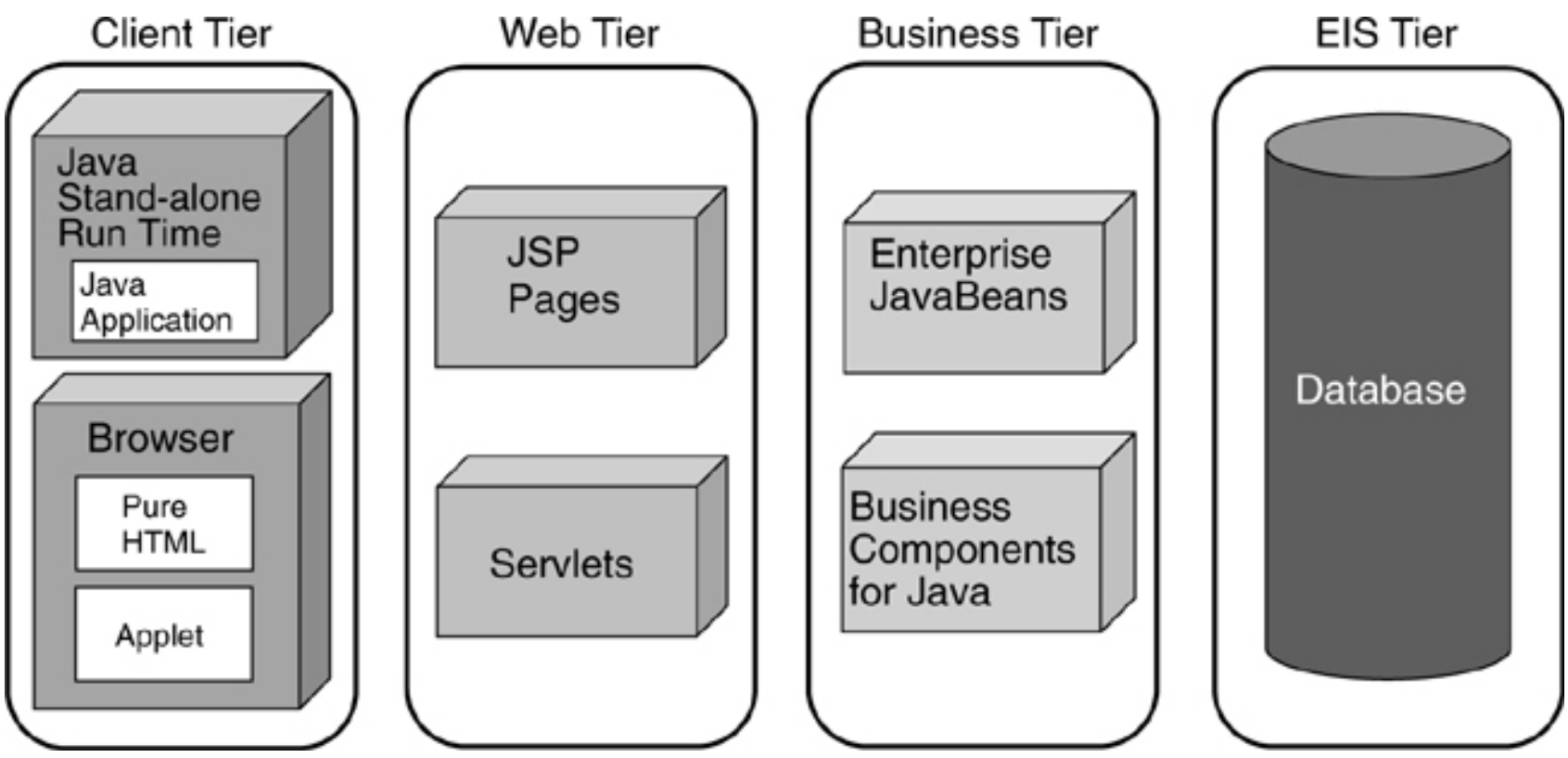

Figure 5.3. A sample J2EE implementation

Note that there is much more to Java than is covered in this section. Refer to the sidebar "Java Program Types" for a brief overview of the different types of Java programs that can be coded.

.NET versus J2EE

There is an ongoing debate as to the relative merits of .NET versus the J2EE platform.

In reality, it is not an "either/or" decision that organizations are making, but a "both/and" decision. Although both are development platforms, the two are not interchangeable.

At a very basic level, .NET is a platform designed to enable development in multiple languages as long as the application is deployed on Windows. On the other hand, J2EE is designed to enable applications to be deployed on any platform as long as they are written in Java.[2] Obviously, an organization may choose to develop different application systems using different methodologies and platforms, depending upon deployment and implementation requirements.

Another difference is that .NET is software that can be purchased from Microsoft. J2EE is a set of specifications (managed by Oracle, formerly Sun Microsystems), each of which dictates how various J2EE functions must operate. IBM's WebSphere Application Server is an example of software that implements J2EE. Oracle makes money from J2EE not only by selling J2EE software, but also by licensing the J2EE specifications to independent software vendors, which then implement software according to the specs.

The bottom line is that both .NET and J2EE can be used to build Web services. A Web service is an application that accepts requests from other systems across the Internet (or an intranet), mediated by lightweight, vendor-neutral communications technologies. Web services enable applications to share data and invoke capabilities from other applications without knowledge of how those other applications were built, what operating system or platform they run on, and what devices are used to access them.

Ruby on Rails

Ruby on Rails is an open-source Web application framework for the Ruby programming language. Ruby on Rails includes tools that make common development tasks easier and is an additional application development framework, similar to but separate from both .NET and J2EE.

Truly, an entire book could be dedicated to the different framework options available for modern application development. But this is a book about database administration, so the cursory overview provided in this section should be sufficient.

Object Orientation and SQL

Many organizations have adopted *object-oriented* (OO) programming standards and languages because of the claimed advantages of the OO development paradigm. The primary advantages of object orientation are faster program development time and

reduced maintenance costs, resulting in a better ROI. Piecing together reusable objects and defining new objects based on similar object classes can dramatically reduce development time and costs.

Object orientation can result in a better ROI.

With benefits like these, it is no wonder that object-oriented programming and development is being embraced by many IT organizations. Historically, one of the biggest problems faced by IT is a large project backlog. In many cases, end users are forced to wait for long periods of time for new applications because the backlog is so great and the talent needed to tackle so many new projects is not available. This backlog can sometimes result in some unsavory phenomena such as business people attempting to build their own applications or purchasing third-party packaged applications (and all of the potential administrative burdens that packages carry). So, it is very clear why the OO siren song lures organizations.

However, because OO and relational databases are not inherently compatible, OO programmers tend to resist using SQL. The set-based nature of SQL is not simple to master and is anathema to the OO techniques practiced by Java and C++ developers. All too often insufficient consideration has been given to the manner in which data is accessed, resulting in poor design and faulty performance.

Object orientation is indeed a political nightmare for those schooled in relational tenets. All too often organizations experience political struggles between OO programming proponents and the data resource management group. The OO crowd espouses programs and components as the center of the universe; the data crowd adheres to normalized, shared data with the RDBMS as the center of the universe.

Thanks to the hype surrounding object orientation, the OO crowd may win many of these battles, but data-centered thinking will eventually win the war. The use of data normalization and shared databases to reduce redundancy provides far too many benefits in the long run for it ever to be abandoned. Data has an existence independent of process, and the OO way of encapsulating data within methods masks data from other processes and is inefficient for sharing data across an organization. If the focus shifts away from data management and sound relational practices, data quality will deteriorate and productivity will decline.

If an OO programming language is to be used against a relational database, you will need to marry the relational world to the OO world. This means your applications will not be object oriented in the true sense of the word because the data will not be encapsulated within the method (that is, the program).

You will need to marry the relational world to the OO world.

There are several techniques that can be used to enable an OO programming language to work with a relational database. Serialization, saving data using a flat file representation of the object, is one approach. Because it can be slow and difficult to use across applications, serialization is not commonly used for persisting object data. A second approach is to use XML, which can be stored natively in many relational database systems. However, XML adds a layer of complexity and requires an additional programming skill set. XML is discussed in more detail later in this chapter.

Probably the most common technique is to deploy an ORM (object-relational mapping) solution. Through ORM, an object's attributes are stored in one or more columns of a relational table. Hibernate is a popular ORM library for Java. NHibernate is an adaptation of Hibernate for the .NET framework. Both Hibernate and NHibernate provide capabilities for mapping objects to a relational database by replacing direct persistence-related database accesses with high-level object-handling functions. Another option is Microsoft LINQ, which stands for Language Integrated Query. LINQ provides a set of .NET framework and language extensions for object-relational mapping.

One additional word of caution here: Many people believe that object-relational databases resolve all of these issues. But an object-relational database is not truly object oriented. The term *object-relational* means basically that the DBMS supports large multimedia data types and gives users the ability to define their own data types and functions—all good things, but not object orientation. So don't get confused over the similarity of the terms.

Types of SQL

SQL, although a single language, comprises multiple types that exhibit different behavioral characteristics and require different development and administration techniques. SQL can be broken down into categories based on execution type, program requirement, and dynamism.

• *SQL can be planned or unplanned.* A planned SQL request is typically embedded into an application program, but it might also exist in a query or reporting tool. At any rate, a planned SQL request is designed and tested for accuracy and efficiency before it is run in a production system. Contrast this with the characteristics of an unplanned SQL request. Unplanned SQL, also called *ad hoc SQL*, is created "on the fly" by end users during the course of business. Most ad hoc queries are created to examine data for patterns and trends that impact business. Unplanned, ad hoc SQL requests can be a significant source of inefficiency and are difficult to tune. How do you tune requests that are constantly written, rewritten, and changed?

SQL can be planned or unplanned, embedded in a program or stand-alone, dynamic or static.

• *SQL can either be embedded in a program or issued stand-alone.* Embedded SQL is contained within an application program, whereas stand-alone SQL is run by itself or within a query, reporting, or OLAP tool.

• *SQL can be dynamic or static.* A dynamic SQL statement is optimized at run time. Depending on the DBMS, a dynamic SQL statement may also be changed at run time. Static SQL, on the other hand, is optimized prior to execution and cannot change without reprogramming. Favor static SQL to minimize the possibility of SQL injection attacks (see the sidebar "SQL Injection").

Programmers must be able to quantify each SQL statement being developed in terms of these three qualities. Every SQL statement exhibits one of these properties for each criterion. For example, a certain SQL statement can be a planned, embedded, static request, or it could be an unplanned, stand-alone, dynamic request. Be sure to use the right type of SQL for the right situation. Table 5.2 outlines situations and the type of SQL that is most useful for that situation. Of course, the information in this table is meant to serve as a broad suggestion only. You should use your knowledge of your environment and the requirements of the user request to arrive at the correct type of SQL solution.

Table 5.2. SQL Usage Considerations

Situation	Execution Type	Program Requirement	Dynamism
Columns and predicates of the SQL statement can change during execution	Planned	Embedded	Dynamic
SQL formulation does not change	Planned	Embedded	Static
Highly concurrent, high-performance transactions	Planned	Embedded	Dynamic or static
Ad hoc one-off queries	Unplanned	Stand-alone	Dynamic
Repeated analytical queries	Planned	Embedded or stand-alone	Dynamic or static
Quick one-time "fix" programs	Unplanned	Embedded or stand-alone	Dynamic or static

SQL Coding for Performance

Developing database application programs requires a good amount of effort to ensure that SQL requests are properly coded for performance. A solid understanding of SQL syntax, database structures, and the programming language is imperative. Let's concentrate first on SQL.

One of the first rules to learn as a database developer is to let SQL, rather than the program logic, do the work. It is much better to filter out unwanted data at the DBMS level than to do so within the program. You'll achieve better efficiency by avoiding the actual movement of data between the DBMS and the program. For example, it is better to add more WHERE clauses to SQL SELECT statements than to simply select all rows and filter the data programmatically.

Let SQL do the work.

To use another example, consider the cost of a multitable join statement. It will be easier to tune, say, a four-table join for efficiency than four independent SQL SELECT statements that are filtered and joined using application logic. Of course, this assumes an optimal physical database and the possibility of having to tweak that design (such as by adding indexes).

The more work the DBMS can do to filter data, the greater the efficiency should be, because less data will need to be moved between the DBMS and the application program as it runs. Of course, there is much more to optimizing and tuning SQL than this short discussion of the matter. More details are covered in Chapter 12, "Application Performance."

Querying XML Data

These days, not all data stored in the database will be relational. XML is gaining popularity for persisting complex data and is frequently used in Web-enabled applications and as a means of data transmission. All of the leading DBMS products provide means of storing and managing XML data.[3]

XML stands for Extensible Markup Language. You may be familiar with HTML, the markup language used to create Web pages. Like HTML, XML is based upon SGML (Standard Generalized Markup Language). SGML is a language for defining markup languages; it was developed and standardized by the International Organization for Standardization (ISO).

XML uses tags to describe the data itself.

Whereas HTML uses tags to describe *how* data appears on a Web page, XML is designed to transport and store data. In other words, XML is somewhat self-describing. It uses tags to describe the *what*, that is, the data itself. The simple syntax of XML makes it easy to process by machine while remaining understandable to people. Once again, let's use HTML as a metaphor to help us understand XML. HTML uses tags to describe the appearance of data on a page. For example the tag "text" would specify that the "text" data should appear in boldface. XML uses tags to

describe the data itself, instead of its appearance. For example, consider the following XML describing a customer address:

Click here to view code image

```
<CUSTOMER>
  <first_name>Craig</first_name>
  <middle_initial>S.</middle_initial>
  <last_name>Mullins</last_name>
  <company_name>Mullins Consulting, Inc.</company_name>
  <street_address>15 Coventry Ct.</street_address>
  <city>Sugar Land</city>
  <state>TX</state>
  <zip_code>77479</zip_code>
  <country>USA</country>
</CUSTOMER>
```

XML is actually a meta-language—that is, a language for defining other markup languages. These languages are collected in dictionaries called Document Type Definitions (DTDs). The DTD stores definitions of tags for specific industries or fields of knowledge. Instead of a DTD, an XML schema could be employed for the same purpose.

When data is stored as XML, it cannot be accessed using standard SQL. Instead, it requires either XQuery or SQL/XML extensions.

XQuery

XQuery is a query and programming language designed to query collections of XML data. XQuery uses XPath expression syntax to address specific parts of an XML document. It supplements this with a SQL-like “FLWOR expression” for performing joins. A FLWOR expression is constructed from the five clauses after which it is named:

- FOR
- LET
- WHERE
- ORDER BY
- RETURN

The XQuery language is not just for querying; it also allows for new XML documents to

be constructed. However, there are no features in XQuery for updating XML documents or databases. It also does not provide full text search capability. Over time, expect these shortcomings to be remedied, though.

In short, XQuery is a programming language that can be used to express arbitrary XML-to-XML data transformations with strong typing and logical/physical data independence.

SQL/XML

SQL/XML is an extension to the SQL standard that specifies SQL-based extensions for using XML in conjunction with SQL. At a high level, it offers an XML data type along with new routines, functions, and XML-to-SQL data type mappings to support accessing and manipulating XML in SQL databases. SQL/XML is developed to be complementary to XQuery.

At the heart of the SQL/XML specification are the functions that allow users to access, modify, and construct XML elements or attributes. Examples of these functions include XMLDOCUMENT (which returns an XML value with a single document node and zero or more nodes as its children), XMLCONCAT (which returns a forest of XML elements generated from a concatenation of two or more elements), XMLELEMENT (which returns an XML element given an element name, an optional collection of attributes, and zero or more arguments that make up the contents of the element), XMLAGG (which returns an XML sequence that contains an item for each non-null value in a set of XML values), among many others.

SQL/XML also defines functions that allow the user to embed XQuery expressions in SQL statements and to convert complex data types. These functions include XMLQUERY and XMLTABLE.

Defining Transactions

A *transaction* is an atomic unit of work with respect to recovery and consistency. A logical transaction performs a complete business process typically on behalf of an online user. It may consist of several steps and may comprise more than one physical transaction. The results of running a transaction will record the effects of a business process—a complete business process. The data in the database must be correct and proper after the transaction executes.

A transaction is an atomic unit of work with respect to recovery and consistency.

When all the steps that make up a specific transaction have been accomplished, a COMMIT is issued. The COMMIT signals that all work since the last COMMIT is correct

and should be externalized to the database. At any point within the transaction, the decision can be made to stop and roll back the effects of all changes since the last COMMIT. When a transaction is rolled back, the data in the database will be restored to the original state before the transaction was started. The DBMS maintains a transaction log (or journal) to track database changes.

Transactions exhibit ACID properties. ACID is an acronym for *atomicity*, *consistency*, *isolation*, and *durability*. Each of these four qualities is necessary for a transaction to be designed correctly.

• *Atomicity* means that a transaction must exhibit "all or nothing" behavior. Either all of the instructions within the transaction happen, or none of them happen. Atomicity preserves the "completeness" of the business process.

• *Consistency* refers to the state of the data both before and after the transaction is executed. A transaction maintains the consistency of the state of the data. In other words, after a transaction is run, all data in the database is "correct."

• *Isolation* means that transactions can run at the same time. Any transactions running in parallel have the illusion that there is no concurrency. In other words, it appears that the system is running only a single transaction at a time. No other concurrent transaction has visibility to the uncommitted database modifications made by any other transactions. To achieve isolation, a locking mechanism is required.

• *Durability* refers to the impact of an outage or failure on a running transaction. A durable transaction will not impact the state of data if the transaction ends abnormally. The data will survive any failures.

Let's use an example to better understand the importance of transactions to database applications. Consider a banking application. Assume that you wish to withdraw $50 from your account with Mega Bank. This "business process" requires a transaction to be executed. You request the money either in person by handing a slip to a bank teller or by using an ATM (automated teller machine). When the bank receives the request, it performs the following tasks, which make up the complete business process. The bank will

1. Check your account to make sure you have the necessary funds to withdraw the requested amount

2. If you do not, deny the request and stop; otherwise continue processing

3. Debit the requested amount from your checking account

4. Produce a receipt for the transaction

5. Deliver the requested amount and the receipt to you

Design transactions that ensure ACID properties.

The transaction that is run to perform the withdrawal must complete all of these steps, or none of these steps, or else one of the parties in the transaction will be dissatisfied. If the bank debits your account but does not give you your money, you will not be satisfied. If the bank gives you the money but does not debit the account, the bank will be unhappy. Only the completion of every one of these steps results in a "complete business process." Database developers must understand the requisite business processes and design transactions that ensure ACID properties.

To summarize, a transaction—when executed alone, on a consistent database—will either complete, producing correct results, or terminate, with no effect. In either case the resulting condition of the database will be a consistent state.

Transaction Guidelines

A transaction should be short in duration because it locks shared resources. Of course, "short" will vary from system to system. A short transaction in a very large system handling multiple thousands of transactions per second will most likely be measured in subseconds.

A transaction should be short in duration.

At any rate, transactions must be designed to remove the human element of "think time" from the equation. When a transaction locks resources, it makes those resources inaccessible to other transactions. Therefore, a good transaction must be designed so that it does not wait for user input in the middle of the processing.

Unit of Work

Unit of work (UOW) is another transaction term that describes a physical transaction. A UOW is a series of instructions and messages that, when executed, guarantee data integrity. So a UOW and a transaction are similar in concept. However, a UOW is not necessarily a complete business process—it can be a subset of the business process, and a group of units of work can constitute a single transaction.

A UOW is a series of instructions and messages that guarantee data integrity.

Each UOW must possess ACID characteristics. In other words, if the transaction were to fail, the state of the data upon failure must be consistent in terms of the business requirements.

Transaction Processing Systems

A transaction processing (TP) system, appropriately enough, facilitates the processing of transactions. Such a system is sometimes referred to as a *transaction server* or a *transaction processing monitor*. Regardless of name, a TP system delivers a scheme to monitor and control the execution of transaction programs. The TP system also provides an API—a mechanism for programs to interact and communicate with the TP server. Examples of TP systems include CICS, IMS/TM, Tuxedo, and Microsoft Transaction Server.

A TP system delivers a scheme to monitor and control the execution of transaction programs.

The TP system provides an environment for developing and executing presentation logic and business logic components. A TP system is useful for mission-critical applications requiring a high volume of concurrent users with minimal downtime. Used properly, a TP system can efficiently control the concurrent execution of many application programs serving large numbers of online users. Another major benefit of some TP systems is their ability to ensure ACID properties across multiple, heterogeneous databases. This is accomplished using a two-phase COMMIT, where the TP system controls the issuance of database commits and ensures their satisfactory completion. If your application requires online access and modification of heterogeneous databases, a TP system is recommended.

A *transaction server* is ideal for building high-performance and reliable distributed applications across heterogeneous environments. A TP system can support the diverse application requirements of front-end e-commerce applications as well as the robust needs of back-office processes. When platform independence is crucial, a TP system can help developers to successfully develop, manage, and deploy online applications that are completely independent of the underlying communications, hardware, and database environment.

Of course, a TP system is not required in order to develop database transactions for every application. The DBMS itself can deliver ACID properties for the data it manages. Yet even in a single-DBMS environment, a TP system can provide development benefits. Take a look at Figure 5.4. It shows the typical application setup: a database server without a TP system. Requests are made by the client through the presentation layer to the database server (DBMS). The client can make data requests directly of the database server, or the client may execute stored procedures to run application logic on the database server.

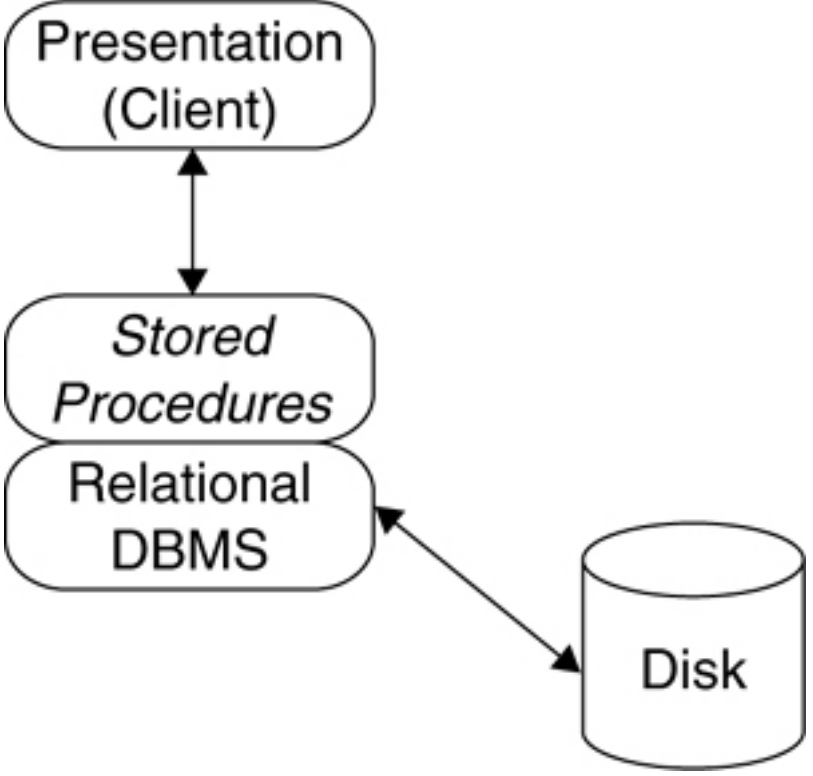

Figure 5.4. Using a database server

Now let's compare Figure 5.4 with Figure 5.5, which shows an application using a TP system. Database requests are run within the TP system, which helps to control the transaction workflow. Furthermore, the TP system enables the application to make requests of multiple heterogeneous databases and to coordinate database modifications.

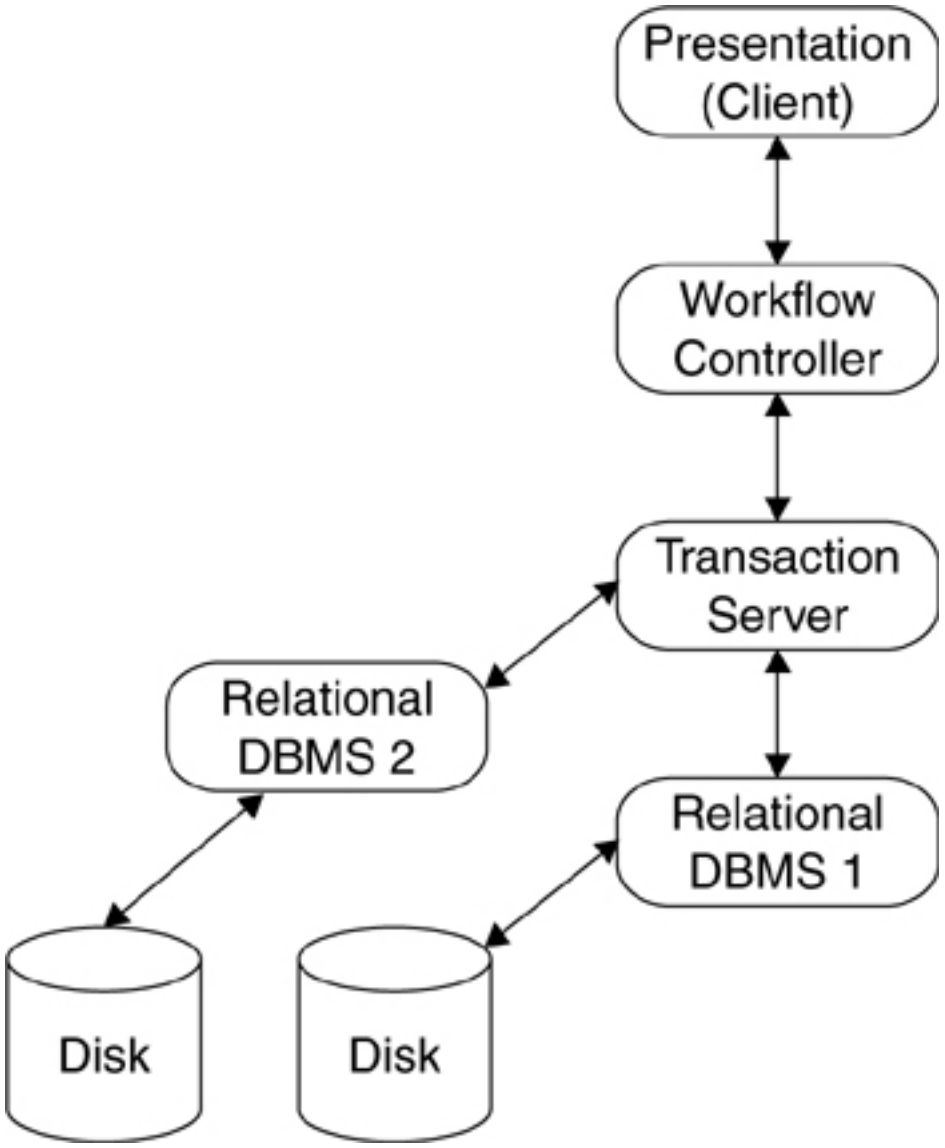

Figure 5.5. Using a transaction server

If your database applications use a TP system, the DBA should work with the system administrator to ensure that the DBMS and the TP system are set up to work with each other in an efficient manner. Furthermore, both the DBA and the SA will need to monitor the interaction between the DBMS and the TP system on an ongoing basis.

Ensure that the DBMS and the TP system are set up to work together in an efficient manner.

Application Servers

A more recent type of middleware for serving database transactions is the *application server*. An application server usually combines the features of a transaction server with additional functionality to assist in building, managing, and distributing database applications. An application server, such as IBM's WebSphere, provides an environment for developing and integrating components into a secure, performance-oriented application.

Other examples of application servers include Zend Server (for PHP-based applications), Base4 Application Server (open source), and TNAPS Application Server (freeware).

There are several advantages of using an application server. When business logic is centralized on an application server, application updates will apply to all users. The risk of having old versions of the application is eliminated. Furthermore, changes to the application configuration, such as moving a database server, can occur centrally and thereby apply to all users at the same time. Additionally, an application server can act as a central point for application security.

With an application server, application changes are made centrally, thereby reducing risk.

Application servers also serve many of the purposes of a transaction server, such as the delivery of transaction support, whereby a unit of work can be made atomic and indivisible.

Locking

Every programmer who has developed database programs understands the potential for concurrency problems. When one program tries to read data that is in the process of being changed by another program, the DBMS must prohibit access until the modification is complete, in order to ensure data integrity. The DBMS uses a *locking* mechanism to enable multiple, concurrent users to access and modify data in the database. By using locks, the DBMS automatically guarantees the integrity of data.

The DBMS locking strategies permit multiple users from multiple environments to access and modify data in the database at the same time.

Locks are used to ensure the integrity of data. When a database resource is locked by one process, another process is not permitted to change the locked data. Locking is necessary to enable the DBMS to facilitate the ACID properties of transaction processing.

Locks are used to ensure the integrity of data.

Data may be locked at different levels within the database. It is the DBA's job to determine the appropriate level of locking for each database object, based on how the data will be accessed and to what extent concurrent users will access the data. Theoretically, database locks can be taken at the following levels:

- Column
- Row
- Page (or block)
- Table
- Tablespace
- Database

The level of locking is known as *lock granularity*. The actual lock granularity levels available will depend on the DBMS in use. Typically the lock granularity is controlled by the DBA when the database object is created, but there may be an overall default lock granularity that is used if none is specified. Nevertheless, it is a good practice for DBAs always to specify the lock granularity when database objects are created.

The lock granularity specification should be based on the needs of the applications and users that will be accessing and changing the data. In general, the smaller the granularity of the lock, the more concurrent access will be allowed, as shown in Figure 5.6. However, the smaller the granularity of the lock, the more resources the DBMS will need to consume to perform locking.

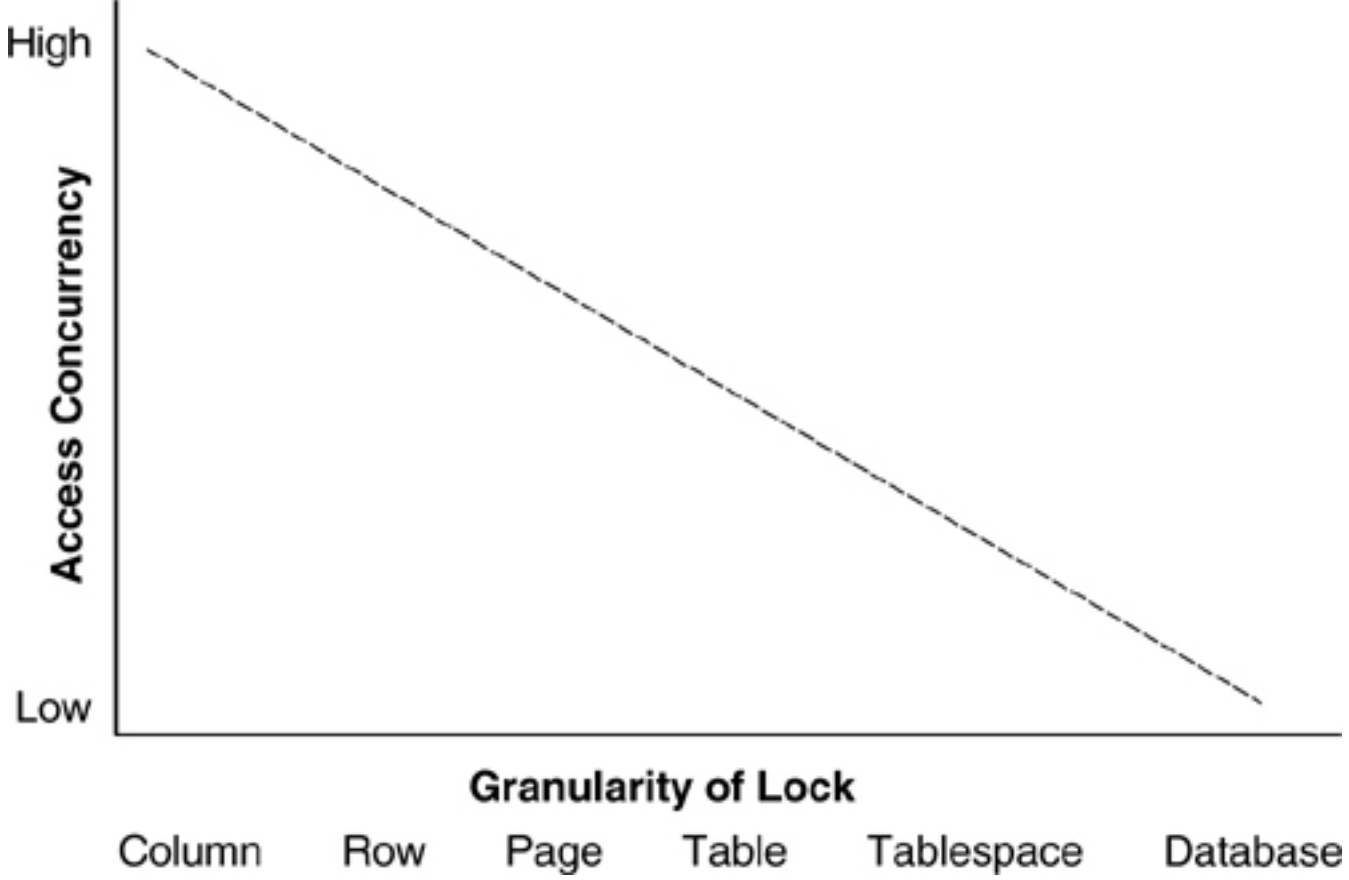

Figure 5.6. Lock granularity and concurrent access

The smallest unit that can conceivably be locked is a single column; the largest lock size is the entire database. Neither of these two options is really practical. Locking individual columns would maximize the ability to concurrently update data, but it would require too much overhead. Conversely, locking the database would be cheap to implement, but it would restrict concurrency to the point of making the database worthless for anybody but a single user.

Most database implementations choose between row and page locking. For many applications, page locking is sufficient. However, applications needing many concurrent update processes may benefit from the smaller granularity of row locking. Very infrequently, when a specific process will be run when no other concurrent access is needed, table locking can be useful. For this reason many DBMS products provide a LOCK TABLE command that can be issued to override the current locking granularity for a database object during the course of a single process or UOW. The DBA must analyze the application processing requirements for each database object to determine the optimal locking granularity.

Most database implementations choose between row and page locking.

Index entries can also be locked, depending on the DBMS and version in use. However, index locking can be a significant impediment to performance. Because index entries are usually quite small, it is not uncommon for locking to block application access—especially when the DBMS locks indexes at the block or page level. Some DBMSs do not require index locks, instead handling integrity by using locks on the data. Remember, there is no data in the index that is not also in a table.

Index locking can be a significant impediment to performance.

The exact nature of locking and the types of locks taken will differ from DBMS to DBMS. This section will cover the basics of locking that are generally applicable to most DBMS products.

Types of Locks

At a very basic level, a DBMS will take a *write lock* when it writes information or a *read lock* when it reads information. A write occurs for INSERT, UPDATE, and DELETE statements. A read occurs for SELECT statements. But to actually accomplish such locking, the typical DBMS will use three basic types of locks: shared locks, exclusive locks, and update locks.

• A *shared lock* is taken by the DBMS when data is read with no intent to update it. If a shared lock has been taken on a row, page, or table, other processes or users are permitted to read the same data. In other words, multiple processes or users can have a shared lock on the same data.

• An *exclusive lock* is taken by the DBMS when data is modified. If an exclusive lock has been taken on a row, page, or table, other processes or users are generally not permitted to read or modify the same data. In other words, multiple processes or users cannot have an exclusive lock on the same data.

• An *update lock* is taken by the DBMS when data must first be read before it is changed or deleted. The update lock indicates that the data may be modified or deleted in the future. If the data is actually modified or deleted, the DBMS will promote the update lock to an exclusive lock. If an update lock has been taken on a row, page, or table, other processes or users generally are permitted to read the data, but not to modify it. So a single process or user can have an update lock while other processes and users have shared locks on the same data. However, multiple processes or users cannot have both an exclusive lock and an update lock, or multiple update locks, on the same data.

Intent Locks

In addition to shared, exclusive, and update locks, the DBMS also will take another type of lock, known as an *intent lock*. Intent locks are placed on higher-level database objects when a user or process takes locks on the data pages or rows. An intent lock stays in place for the life of the lower-level locks.

An intent lock stays in place for the life of the lower-level locks.

For example, consider a table created with row-level locking. When a process locks the

row, an intent lock is taken on the table. Intent locks are used primarily to ensure that one process cannot take locks on a table, or pages in the table, that would conflict with the locking of another process. For example, if a user was holding an exclusive row lock and another user wished to take out an exclusive table lock on the table containing the row, the intent lock held on the table by the first user would ensure that its row lock would not be overlooked by the lock manager.

Lock Time-outs

When data is locked by one process, other processes must wait for the lock to be released before processing the data. A lock that is held for a long time has the potential to severely degrade performance because other processes must wait until the lock is released and the data becomes available. Furthermore, if the application is designed improperly or has a bug, the blocking lock may not be released until the program fails or the DBA intervenes.

The locking mechanism of the DBMS prevents processes from waiting forever for a lock to be released by timing out. Each DBMS provides a parameter to set a *lock time-out value*. Depending on the DBMS, the lock time-out value might be set at the DBMS level, the process level, or the connection level. Regardless of the level, after a process waits for the predetermined amount of time for a lock to be granted, the process will receive an error message informing it that the time-out period has been exceeded. Such an approach assumes that a problem occurs after a certain amount of time is spent waiting for a lock. Time-outs prevent a process from waiting indefinitely for locks —the rationale being that it is better for a process to give up and release its locks than to continue to wait and perhaps block other processes from running.

It is usually a good practice for programs to retry an operation when a lock time-out error is received. If multiple lock time-outs occur for the same operation, the program should log the problem and inform the user that it cannot proceed.

Deadlocks

Another locking problem that can occur is deadlocking. A *deadlock* occurs when concurrent processes are competing for locks. Figure 5.7 shows a deadlock situation. Process A holds a lock on row 3 and is requesting a lock on row 7; process B holds a lock on row 7 and is requesting a lock on row 3.

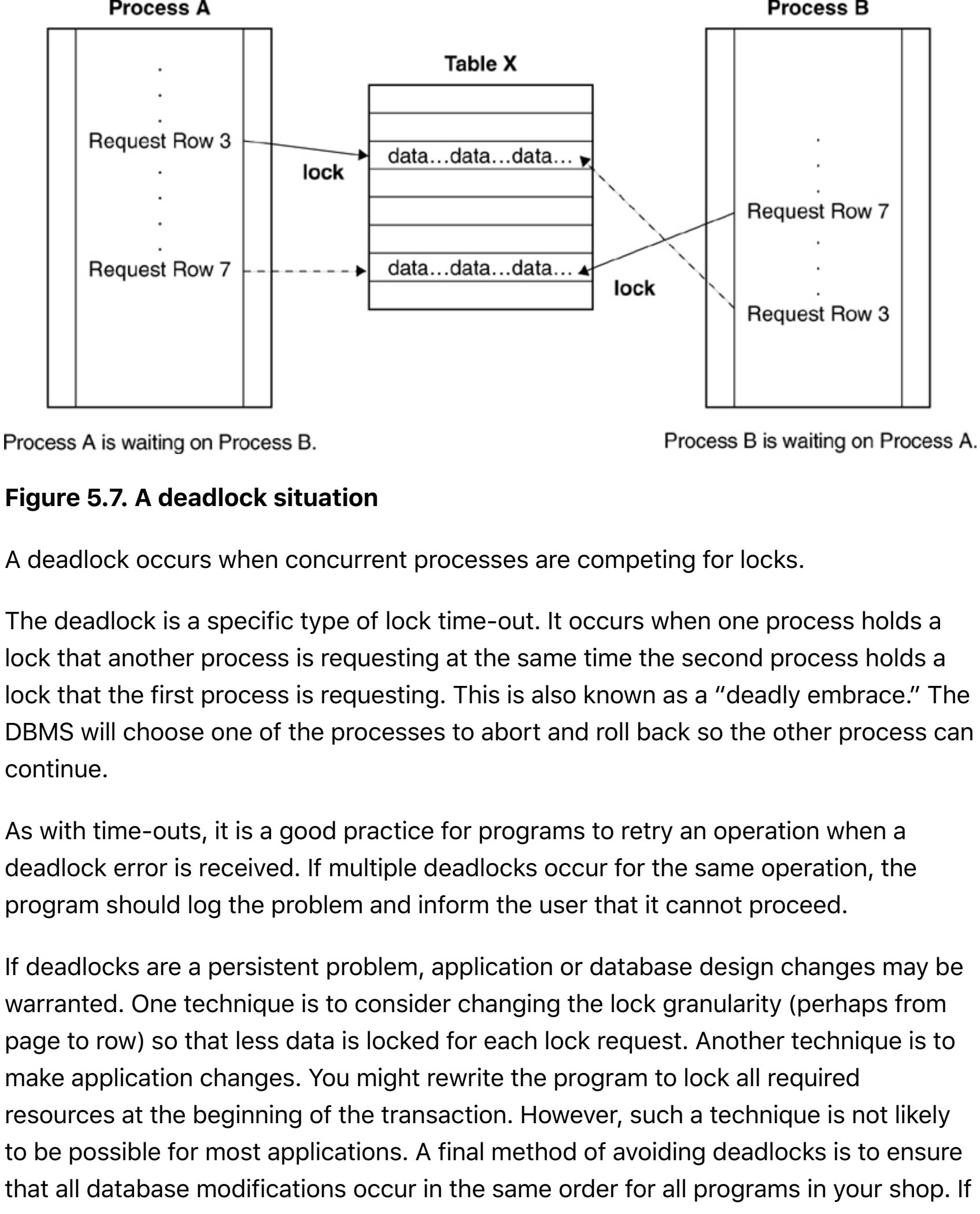

Figure 5.7. A deadlock situation

A deadlock occurs when concurrent processes are competing for locks.

The deadlock is a specific type of lock time-out. It occurs when one process holds a lock that another process is requesting at the same time the second process holds a lock that the first process is requesting. This is also known as a “deadly embrace.” The DBMS will choose one of the processes to abort and roll back so the other process can continue.

As with time-outs, it is a good practice for programs to retry an operation when a deadlock error is received. If multiple deadlocks occur for the same operation, the program should log the problem and inform the user that it cannot proceed.

If deadlocks are a persistent problem, application or database design changes may be warranted. One technique is to consider changing the lock granularity (perhaps from page to row) so that less data is locked for each lock request. Another technique is to make application changes. You might rewrite the program to lock all required resources at the beginning of the transaction. However, such a technique is not likely to be possible for most applications. A final method of avoiding deadlocks is to ensure that all database modifications occur in the same order for all programs in your shop. If the updates in every program are sequenced, deadlocks can be minimized or eliminated. Perhaps you’ll find a logical business order for updates that is reasonable to use for applications. If you don’t, consider using an arbitrary policy such as modification in alphabetical order.

Lock Duration

Lock duration refers to the length of time that a lock is held by the DBMS. The longer a

lock is held on a database resource, the longer a concurrent lock request must wait to be taken. As lock durations increase, so does the likelihood of lockout time-outs.

Lock duration refers to the length of time that a lock is held by the DBMS.

Each DBMS provides parameters that can be set to impact the duration of a lock. Typically these parameters are set at the program, transaction, or SQL statement level. In general, there are two parameters that affect lock duration: isolation level and acquire/release specification.

Isolation Level

The *isolation level* specifies the locking behavior for a transaction or statement. Standard SQL defines four isolation levels that can be set using the SET TRANSACTION ISOLATION LEVEL statement:

- UNCOMMITTED READ
- COMMITTED READ
- REPEATABLE READ
- SERIALIZABLE

The preceding list progresses from lowest to highest isolation. The higher the isolation level, the stricter the locking protocol becomes. The lower the isolation level, the shorter the lock duration will be. Additionally, each higher isolation level is a superset of the lower levels. Let's briefly examine each of the standard isolation levels.

The isolation level specifies the locking behavior for a transaction or statement.

Specifying *UNCOMMITTED READ* isolation implements read-through locks and is sometimes referred to as a *dirty read*. It applies to read operations only. With this isolation level, data may be read that never actually exists in the database, because the transaction can read data that has been changed by another process but is not yet committed. UNCOMMITTED READ isolation provides the highest-level availability and concurrency of the isolation levels, but the worst degree of data integrity. It should be used only when data integrity problems can be tolerated. Certain types of applications, such as those using analytical queries, estimates, and averages, are often candidates for UNCOMMITTED READ locking. A dirty read can cause duplicate rows to be returned where none exist, or no rows to be returned when one (or more) actually exists. When choosing UNCOMMITTED READ isolation, the programmer and DBA must ensure that these types of problems are acceptable for the application. Refer to the sidebar "Dirty Read Scenarios" for additional guidance on when to consider using

UNCOMMITTED READ isolation.

COMMITTED READ isolation, also called *cursor stability*, provides more integrity than UNCOMMITTED READ isolation. When READ COMMITTED isolation is specified, the transaction will never read data that is not yet committed; it will only COMMITTED READ data.

REPEATABLE READ isolation places a further restriction on reads, namely, the assurance that the same data can be accessed multiple times during the course of the transaction without its value changing. The lower isolation levels (UNCOMMITTED READ and COMMITTED READ) permit the underlying data to change if it is accessed more than once. Use REPEATABLE READ isolation only when data can be read multiple times during the course of the transaction and the data values must be consistent.

Finally, *SERIALIZABLE* isolation provides the greatest integrity. SERIALIZABLE isolation removes the possibility of phantoms. A phantom occurs when the transaction opens a cursor that retrieves data and another process subsequently inserts a value that would satisfy the request and should be in the results set. For example, consider the following situation:

• Transaction 1 opens a cursor and reads account information, keeping a running sum of the total balance for the selected accounts.

• Transaction 2 inserts a new account that falls within the range of accounts being processed by Transaction 1, but the insert occurs after Transaction 1 has passed the new account.

• Transaction 2 COMMITs the insert.

• Transaction 1 runs a query to sum the values to check the accuracy of the running total. However, the totals will not match.

SERIALIZABLE isolation provides the greatest integrity.

SERIALIZABLE isolation eliminates this problem. Phantoms can occur for lower isolation levels, but not when the isolation level is SERIALIZABLE.

Most DBMS products support the specification of isolation level at the program or transaction level, as well as at the SQL statement level.

Keep in mind that your DBMS may not implement all of these isolation levels, or it may refer to them by other names. Be sure you understand the isolation supported by each DBMS you use and its impact on application behavior and lock duration.

Acquire/Release Specification

An additional parameter that impacts lock duration is the treatment of intent locks. Regular transaction locks are taken as data is accessed and modified. However, some DBMS products provide methods to control when intent locks are taken. Intent locks can be acquired either immediately when the transaction is requested or iteratively as needed while the transaction executes. Furthermore, intent locks can be released when the transaction completes or when each intent lock is no longer required for a unit of work.

If the DBMS supports different options for the acquisition and release of intent locks, the parameter will be specified at the transaction or program level.

Skipping Locked Rows

An additional locking option, available in some DBMSs,[4] is the ability to skip locked data. If you code a parameter specifying SKIP LOCKED DATA on certain SQL statements, any data that is locked will simply be skipped over instead of the DBMS waiting for the lock to be released.

This option should be used only sparingly and only with a full understanding of its impact. When you tell the DBMS to skip locked data, that data is not accessed and will not be available to your program. The benefit, of course, is improved performance because you will not incur any lock wait time; however, it comes at the cost of not accessing the locked data.

You should use this feature sparingly and with extreme caution. Before skipping locked data, make sure that you completely understand exactly what you are telling the DBMS to do. It is very easy to misuse this feature and wind up reading less data than you want.

Consider using this option in certain test environments, in a data warehouse where data is read only, and possibly even in production under the proper conditions. For example, perhaps you have a program that needs to read from a table such as a queue to get a next number. If it is not imperative that the numbers be sequential, skipping locked data can eliminate bottlenecks by skipping any locked rows/pages to get data off of the queue.

Lock Escalation

If processing causes the DBMS to hold too many locks, lock escalation can occur. *Lock escalation* is the process of increasing the lock granularity for a process or program. When locks are escalated, more data is locked, but fewer actual locks are required. An

example of escalating a lock would be moving from page locks to table locks. You can see where this would minimize the number of locks the DBMS needs to track—multiple page locks for a table can be converted into a single lock on the entire table. Of course, this impacts concurrent access because the process locks the entire table, making it inaccessible to other processes.

Lock escalation is the process of increasing the lock granularity for a process or program.

The DBMS kicks off lock escalation based on preset thresholds. Typically, the DBMS will provide system parameters that can be set to customize the actual manner in which the DBMS escalates locks, or to turn off lock escalation. Also, the DBMS will provide DDL parameters for database objects to indicate on an object-by-object basis whether escalation should occur.

Some DBMSs, such as DB2 and Microsoft SQL Server, provide the capability to escalate locks, whereas others, such as Oracle, do not. However, both DB2 and Microsoft SQL Server can escalate from page locks to table locks or from row locks to table locks. Neither allows escalation from row locks to table locks.

Programming Techniques to Minimize Locking Problems

We have learned that locking is required to ensure data integrity. If application programs are not designed with database locking in mind, though, problems can arise. Application developers must understand the impact of locking on the performance and availability of their applications. If locks are held too long, time-outs will make data less available. If applications request locks in a disorganized manner, deadlocks can occur, causing further availability problems.

Standardize the sequence of updates within all programs.

Fortunately, though, there are development techniques that can be applied to minimize locking problems. One such technique is to standardize the sequence of updates within all programs. When the sequence of updates is the same for all programs, deadlocks should not occur.

Another programming technique is to save all data modification requests until the end of the transaction. The later modifications occur in a transaction, the shorter the lock duration will be. From a logical perspective, it really does not matter where a modification occurs within a transaction, as long as all of the appropriate modifications occur within the same transaction. However, most developers feel more comfortable scattering the data modification logic throughout the transaction in a pattern that matches their concept of the processes in the transaction. Grouping modifications

such as INSERT, UPDATE, and DELETE statements and issuing them near the end of the transaction can improve concurrency because resources are locked for shorter durations.

Locking Summary

Database locking is a complex subject with more details than we have covered in this section. Each DBMS performs locking differently, and you will need to study the behavior of each DBMS you use to determine how best to set locking granularity and isolation levels, and to program to minimize time-outs and deadlocks.

Batch Processing

Most of the discussion in this chapter has centered around transaction processing, which is usually assumed to be for online processes. However, many database programs are designed to run as batch jobs with no online interaction required. DBAs must be aware of the special needs of *batch database programs*.

The first design concern for batch database programs is to ensure that database COMMITs are issued within the program. Except for very trivial programs that access small amounts of data, database COMMITs should be issued periodically within a batch program to release locks. Failure to do so can cause problems such as a reduction in availability for concurrent programs because a large number of locks are being held, or a large disruption if the batch program aborts, because all the database modifications must be rolled back.

Ensure that database COMMITs are issued within the program.

Additionally, if a batch program with no COMMITs fails, all of the work that is rolled back must be performed again when the problem is resolved and the batch program is resubmitted for processing. A batch program with COMMITs must be designed for restartability: The batch program must keep track of its progress by recording the last successful COMMIT and including logic to reposition all cursors to that point in the program. When the program is restarted, it must check to see if it needs to reposition and, if so, execute the repositioning logic before progressing.

Another problem that occurs frequently with batch database program development is a tendency for developers to think in terms of file processing, rather than database processing. This is especially true for mainframe COBOL programmers who have never worked with database systems. Each developer must be trained in the skills of database programming, including SQL skills, set-at-at-time processing, and database optimization. The responsibility for assuring that developers have these skills quite often falls on the DBA.

Finally, batch programs typically are scheduled to run at predetermined times. The DBA should assist in batch database job scheduling to help minimize the load on the DBMS. Batch jobs that are long running and resource consuming should be scheduled during off-peak online transaction processing hours.

Summary

Application design and development is the job of systems analysts and application programmers. However, the DBA must be involved in the process when programs are being written to access databases. Special skills are required that can be difficult to master. The DBA must first understand these skills and then work to transfer the knowledge to developers. This is a continual job because new programmers are constantly being hired—each with a different level of skill and degree of database experience. Furthermore, DBMS products are constantly changing, resulting in additional development options and features that need to be mastered.

Review

1. Describe what the acronym *ACID* means and define each component.

2. What is ORM and why would it be necessary for application development?

3. Why is locking required to assure data integrity?

4. Describe the difference between CURSOR STABILITY and REPEATABLE READ isolation levels.

5. Under what circumstance should an isolation level of UNCOMMITTED READ be considered?

6. Describe two application design techniques to minimize the impact of locking on application performance.

7. What does *relational closure* mean, and what is its significance in application design?

8. Describe, at a high level, what is required to embed SQL into an application program written in a programming language like C or Visual Basic.

9. What is the difference between a lock time-out and a deadlock?

10. What programming techniques can be used to minimize deadlocks and why?

Bonus Question

Why might the order of database modifications within a transaction impact deadlocks?

Suggested Reading

Anagol-Subbarao, Anjali. *J2EE Web Services on BEA WebLogic*. Upper Saddle River, NJ: Prentice Hall (2005). ISBN 0-13-143072-6.

Applequist, Daniel K. *XML and SQL: Developing Web Applications*. Boston, MA: Addison-Wesley (2002). ISBN 0-201-65796-1.

Bales, Donald. *Java Programming with Oracle JDBC*. Sebastopol, CA: O'Reilly (2002). ISBN 0-596-00088-X.

Barcia, Roland, et al. *Persistence in the Enterprise: A Guide to Persistence Technologies*. Upper Saddle River, NJ: IBM Press (2008). ISBN 978-0-13-158756-4.

Beighley, Lynn, and Michael Morrison. *Head First PHP & MySQL*. Sebastopol, CA: O'Reilly (2009). ISBN 978-0-596-00630-3.

Bernstein, Philip A., and Eric Newcomer. *Principles of Transaction Processing*. San Francisco, CA: Morgan Kaufmann (1997). ISBN 1-55860-415-4.

Ceri, Stefano, et al. *Designing Data-Intensive Web Applications*. San Francisco, CA: Morgan Kaufmann (2003). ISBN 1-55860-190-2.

Date, C. J., with Hugh Darwen. *A Guide to the SQL Standard*. 4th ed. Reading, MA: Addison-Wesley (1997). ISBN 0-201-96426-0.

Dix, Paul. *Service-Oriented Design with Ruby and Rails*. Boston, MA: Addison-Wesley (2010). ISBN 978-0-321-65936-1.

Donahoo, Michael J., and Gregory D. Speegle. *SQL: Practical Guide for Developers*. San Francisco, CA: Morgan Kaufmann (2005). ISBN 978-0-12-220531-6.

Fronckowiak, John W. *Teach Yourself OLE DB and ADO in 21 Days*. Indianapolis, IN: SAMS Publishing (1997). ISBN 0-672-31083-X.

Garvin, Curtis, and Steve Eckols. *DB2 for the COBOL Programmer Part 1*. 2nd ed. Fresno, CA: Mike Murach & Associates (1999). ISBN 1-890774-16-2.

Garvin, Curtis, and Anne Prince. *DB2 for the COBOL Programmer Part 2*. 2nd ed. Fresno, CA: Mike Murach & Associates (1999). ISBN 1-890774-03-0.

Gray, Jim, and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann (1993). ISBN 1-55860-190-2.

Geiger, Kyle. *Inside ODBC*. Redmond, WA: Microsoft Press (1995). ISBN 1-55615-815-7.

Gulutzan, Peter, and Trudy Pelzer. *SQL-99 Complete, Really*. Lawrence, KS: R&D Books (1999). ISBN 0-87930-568-1.

Harrington, Jan L. *SQL Clearly Explained*. 3rd ed. Burlington, MA: Morgan Kaufmann (2010). ISBN 978-0-12-375697-8.

Jennings, Roger. *Database Developer's Guide with Visual Basic 6*. Indianapolis, IN: SAMS Publishing (1999). ISBN 0-672-31063-5.

Jepson, Brian. *Java Database Programming*. New York, NY: John Wiley & Sons (1997). ISBN 0-471-16518-2.

Kaute, Pierre Henri, Tobin Harris, Christian Bauer, and Gavin King. *NHibernate in Action*. Greenwich, CT: Manning Publications (2009). ISBN 978-1-932394-92-4.

Kline, Kevin, with Daniel Kline. *SQL in a Nutshell*. 3rd ed. Sebastopol, CA: O'Reilly (2009). ISBN 978-0-596-51884-4.

Lewis, Philip M., Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing*. Boston, MA: Addison-Wesley (2002). ISBN 0-201-70872-8.

Loosley, Chris, and Frank Douglas. *High-Performance Client/Server*. New York, NY: John Wiley & Sons (1998). ISBN 0-471-16269-8.

Marguerie, Fabrice, Steve Eichert, and Jim Wooley. *LINQ in Action*. Greenwich, CT: Manning Publications (2008). ISBN 978-1-933988-16-0.

Melton, Jim. *Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM*. San Francisco, CA: Morgan Kaufmann (1998). ISBN 1-55860-461-8.

Patrick, Tim. *ADO.NET 4: Step by Step*. Sebastopol, CA: Microsoft Press (2010). ISBN 978-0-7356-3888-4.

Price, Jason. *Java Programming with Oracle SQLJ*. Sebastopol, CA: O'Reilly (2001). ISBN 0-596-00087-1.

Pugh, Eric, and Joseph D. Gradecki. *Professional Hibernate*. Indianapolis, IN: Wrox (2004). ISBN 0-7645-7677-1.

Reese, George. *Java Database Best Practices*. Sebastopol, CA: O'Reilly (2003). ISBN 0-596-00522-9.

Rinehart, Martin. *Java Database Development*. Berkeley, CA: McGraw-Hill (1998). ISBN 0-07-882356-0.

Sceppa, David. *Programming Microsoft ADO.NET*. Sebastopol, CA: Microsoft Press (2012). ISBN 978-0-7356-4801-2.

Syverson, Bryan. *Murach's SQL for SQL Server*. Fresno, CA: Mike Murach & Associates (2002). ISBN 1-890774-16-2.

Walmsley, Priscilla. *XQuery*. Sebastopol, CA: O'Reilly (2007). ISBN 978-0-596-00634-1.

Yank, Kevin. *Build Your Own Database Driven Web Site Using PHP and MySQL*. 3rd ed. Collingwood, VIC, Australia: SitePoint (2004). ISBN 0-9752402-1-8.