0471269239_01.qxd 1/23/03 9:00 AM Page 1

CHAPTER 1

An Overview of MySQL

In this chapter, we explain why you might choose to use a database system with your software. We also provide an overview of the MySQL database server and the Connector/J JDBC driver.

For many years, large corporations have enjoyed the ability to deploy relational database management systems (RDBMSs) across their enterprise. Companies have used these systems to collect vast amounts of data that serve as the "fuel" for numerous applications that create useful business information.

Until recently, RDBMS technology has been out of reach for small businesses and individuals. Widely used RDBMS systems such as Oracle and DB2 require complex, expensive hardware. License fees for these systems are in the tens to hundreds of thousands of dollars for each installation. Businesses must also hire and retain staff with specialized skill sets to maintain and develop these systems. Smaller enterprises have relied on systems like Microsoft Access and FoxPro to maintain their corporate data.

Early on, during the explosive growth of the Internet, open source database systems like mSQL, Postgres (now PostgreSQL), and MySQL became available for use. Over a relatively short amount of time, the developers of these systems have provided a large subset of the functionality provided by the expensive commercial database systems. These open source database systems also run on less-expensive commodity hardware, and have proven in many cases to be easier to develop for and maintain than their commercial counterparts.

Finally, smaller businesses and individuals have access to the same powerful level of software tools that large corporations have had access to for over a decade.

Why Use an RDBMS?

Almost every piece of software that has been developed needs to *persist* or store data. Once data has been persisted, it is natural to assume that this data needs to be retrieved, changed, searched, and analyzed.

You have many options for data persistence in your software, from rolling your own code, to creating libraries that access flat files, to using full-blown RDBMS systems. Factors to consider when choosing a persistence strategy include whether you need multiuser access, how you will manage storage requirements, whether you need transactional integrity, and whether the users of your software need ad hoc query capability. RDBMSs offer all of this functionality.

Multiuser Access

Many programs use flat files to store data. Flat files are simple to create and change. The files can be used by many tools, especially if they are in comma- or tab-delimited formats. A large selection of built-in and third-party libraries is available for dealing with flat files in Java. The java.util.Properties class included with the Java Development Kit is one example.

Flat file systems can quickly become untenable when multiple users require simultaneous access to the data. To prevent corrupting the data in your file, you must lock the file during changes, and perhaps even during reads. While a file is locked, it cannot be accessed by other users. When the file becomes larger and the number of users increases, this leads to a large bottleneck because the file remains locked most of the time—your users are forced to wait until they can have exclusive access to the data.

RDBMSs avoid this situation by employing a number of locking strategies at varying granularities. Rather than using a single lock, the database system can lock an individual table, an individual page (a unit of storage in the database, usually covering more than one row), or an individual row. This increases throughput when multiple users are attempting to access your data, which is a common requirement in Web-based or enterprise-wide applications.

Storage Transparency

If you use flat files in your software, you are also responsible for managing their storage on disk. You have to figure out where and how to store the data, and

0471269239_01.qxd 1/23/03 9:00 AM Page 3

every time the location or layout of the files changes, you are required to change your software. Once the datasets your software is storing become numerous or large, the storage management process becomes cumbersome.

Using a database system gives you “storage transparency.” Your software does not care where and how the data is stored. The data can even be stored on some other computer and accessed via networking protocols.

Transactions

When you have more than one user accessing and changing your data, you want to make these changes *transactional*. Transactions group operations on your data into units of work that meet the *ACID* test. The ACID test concept is best illustrated with a commonly used example from the banking industry.

Jack and Jill share a joint checking account with a balance of $1000. They are both performing various operations, such as deposits, withdrawals, and transfers, on the account. Let’s see how the four aspects of the ACID test come into play:

- **Atomicity:** All changes made during a transaction are made successfully, or in the case of failure, none are made. If any operation fails during the transaction, then the entire transaction is rolled back, leaving your data in the state it was before the transaction was started. For example, suppose Jack is making a transfer of $500 from his checking account to a savings account. Sometime between the withdrawal of the $500 from the checking account and the deposit of $500 to the savings account, the software running the banking system crashes. Jack’s $500 has disappeared! With atomicity, either the entire transfer would have happened, or none of it would have happened, leaving Jack a much happier customer than he is now.
- **Consistency:** All operations transform the database from one consistent state to another consistent state. Consistency is defined by how the database schema is designed and whether integrity constraints such as foreign keys are used. The database management system is responsible for ensuring that transactions do not violate the database schema or integrity constraints. For example, the bank’s database developers have declared in the database schema that the balance of an account cannot be empty, or “null.” If any transaction attempts to set the balance to an empty value, the transaction will be aborted and any changes rolled back.
- **Isolation:** A transaction’s changes are not made visible to other transactions until they are committed under the atomicity rule described earlier. This is best demonstrated by what happens when month-end reports are generated. Let’s say that Jack is performing the transfer transaction outlined in the atomicity example, and at the same time you are generating his

0471269239_01.qxd 1/23/03 9:00 AM Page 4

monthly statement. Without isolation, the monthly statement might show the withdrawal from the checking account but not the deposit into the savings account. This discrepancy would make it impossible for Jack or the bank to balance their books.

- **Durability:** Once completed, a transaction's changes are never lost through system or hardware crashes. If Jill has paid for $50 worth of groceries with her debit card at the grocery store and the transaction succeeds, even if the database software crashes immediately after the transaction competes, it won't forget that her checking account balance is $50 lower.

Until recently, MySQL did not comply with all components of the ACID test. However, with the new BDB and InnoDB table types (supported in MySQL 3.23 and MySQL 4.0), MySQL can now pass the ACID test.

Not all software requires the robustness (or the associated overhead) of transaction semantics. MySQL is one of the only databases that enable you to decide what level of robustness you need on a table-by-table basis. This becomes important when you are trying to maximize performance, especially when much of the data is read-only (such as in a product catalog).

Searching, Modifying, and Analyzing Data

Any time you store a significant amount of data with your software, your users want to search, modify, and analyze the data you have stored. If you are using flat files, you most likely have to develop this functionality yourself.

As your data stored in flat files takes up more and more space, it takes longer and longer to search. A common solution to this problem is to create an index for your data. Indexes are basically shortcuts to finding a particular piece of data, usually using some sort of key. If you need to develop indexing functionality yourself, you have to learn about data structures, such as hashes and B-trees, and how to store these indexes alongside your data. In addition, you must learn how to implement the index in your software. If you use an RDBMS, you can tell the database system what data you think people will search on, and it does all of the fancy indexing for you.

Users of your software also want to retrieve, modify, and analyze the data you have stored. They expect that your system knows how to compute such values as sums, averages, minimums, and maximums to be used for updating related data or analyzing existing data. They expect that your software will be able to sort the data or group the data by similar attributes. All of this functionality requires you to implement numerous functions and algorithms. If you use an RDMBS, all of these features are built in.

0471269239_01.qxd 1/23/03 9:00 AM Page 5

Ad Hoc Queries

It is likely that your software will need to retrieve stored data using arbitrary parameters, otherwise known as *ad hoc* queries. This becomes difficult with flat files because they are not self-describing, and every file layout is different. You also need to consider how you are going to read the data for these queries from your persistent storage mechanism.

Many RDBMSs use SQL (Structured Query Language) for manipulating data. SQL is a declarative language in that you declare *what* data you want, not the procedure for *how* to get it. SQL is also an accepted and widely used standard, so a large set of tools are available (JDBC and Enterprise Java Beans, among them) to help you work with it.

After outlining all of the benefits of an RDBMS, I hope you are ready to consider using one for your software projects. The next question to ask is "Why choose MySQL?"

Why Choose MySQL?

As was the case with many other open source projects, MySQL was first created by someone who needed a better tool to get a specific job done. Monty Widenius and David Axmark started out with another open source project (MSQL), but found that it lacked some features that they needed. They decided to develop their own database system that met their specific requirements. They started building MySQL by using some low-level database storage code they had already developed for other projects and layered a multithreaded server, SQL parser, and client-server protocol on top. They also structured the API for MySQL to appear very similar to MSQL in order to make it easier for developers to port their MSQL-based software to MySQL.

MySQL was eventually released in source-code form, under a proprietary license. Eventually, this license was changed to the GNU General Public License (GPL), which in most cases allows the software to be used without license cost. However, in certain situations you must purchase a commercial license. The exact terms of the license are available in the documentation that ships with MySQL or on the Web at www.mysql.com. Commercial support is also available for those who need it from MySQL-AB, the company that was created by Monty and David to support the continued development of the MySQL software.

The requirements that Monty and David originally had for MySQL were that it be as fast as possible, while still being stable, simple to use, and able to meet the needs of the majority of database developers. Even today, feature requests for future MySQL development are weighed carefully against these original

requirements, and are implemented only when and if the original requirements can be met as much as possible.

Over the years, MySQL has evolved into an RDBMS that has the following core features:

- **Portability:** MySQL runs on almost every flavor of Unix, as well as Windows and MacOS X. You can obtain binaries or source code for the MySQL server as well as the tools that access it. More ports of the software become available every day. It is almost a given that MySQL will run on whatever operating system you have available.
- **Speed:** Using techniques such as efficient indexing mechanisms, in-memory temporary tables, and highly optimized join algorithms, MySQL executes most queries much faster than most other database systems.
- **Scalability:** Because of its modularity and its flexibility in configuration, MySQL can run in systems varying in size from embedded systems to large multiprocessor Unix servers hosting databases with tens of millions of records. This scalability also allows you to run a copy of MySQL on a developer-class machine, and later use the same database system on a larger machine in production. Because it is multithreaded, MySQL efficiently utilizes resources for multiple users, compared to other database servers that start full-fledged processes for each user. It is not uncommon to hear of MySQL installations supporting thousands of concurrent users.
- **Flexibility:** MySQL lets you choose the table types you need to meet your software's requirements, ranging from in-memory heap tables, fast on-disk MyISAM tables, merge tables that group together other sets of tables to form larger "virtual" tables, and transaction-safe tables such as InnoDB. MySQL is also very tunable and includes many parameters that can be changed to increase performance for a given solution. However, MySQL comes with sensible defaults for these parameters, and many users never have to tune MySQL to reach a performance they are happy with.
- **Ease of use:** MySQL is easy to install and administer. While other database systems require special knowledge and training, not to mention special operating system configurations, MySQL can be installed in less than 10 minutes if you've done it before. Even if you are a newcomer, you should be able to install MySQL in under an hour. Once it's installed, MySQL requires little maintenance and administration other than adding or changing user permissions and creating or removing databases.
- **Fine-grained security model:** You can restrict users' rights from an entire database down to the column level based on login name, password, and the hostname that users are connecting from. This allows you to create secure systems by partitioning responsibilities and capabilities of different users and applications to prevent unauthorized modification or retrieval of data.

0471269239_01.qxd 1/23/03 9:00 AM Page 7

- **Access from other languages/systems:** There are libraries and APIs for connecting to MySQL from Java (the focus of this book), C/C++, Perl, PHP, ODBC (Microsoft Windows applications), TCL, Eiffel, and Lisp. Because of this, a whole set of tools has appeared surrounding the use of MySQL from these languages and systems.

As you can see, MySQL is a flexible and capable RDBMS that has a rich feature set, performs well on the majority of queries, and has a large support base for access from many different languages. This book focuses on using MySQL with JDBC, which is what we talk about next.

MySQL and JDBC

Many developers choose to implement software using Sun's Java technology because of the support Java has for standard Internet concepts such as Web sites, e-mail, and networking. This is the very reason I started to investigate using Java with MySQL in 1994.

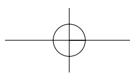
Sun created a standardized interface to databases from Java called Java Database Connectivity (JDBC). Early in 1994, I was interested in connecting a Java application I was about to develop with the then-new MySQL database system using JDBC.

At the time, a rudimentary JDBC driver developed by GWE Technologies existed for MySQL. However, it was missing many features that I required for my project. Because many of the features that I needed would have been difficult to implement in the original MySQL driver, I decided to see if I could implement one myself, more as a tutorial than anything else.

After a few weeks of work, I had something that met most of my needs. Through correspondence with other Java developers on the MySQL mailing list, I found that others had a need for a JDBC driver to use with MySQL, and that they required many of the features I had just implemented. Not knowing what would happen, I wrote about the driver I had developed and allowed people to use it. From that small project, the JDBC driver known as MM.MySQL was born.

Over the years, through many hundreds of e-mails from users around the world, chronicling bugs and interoperability issues with development tools and application servers, MM.MySQL was fixed and tuned and eventually stabilized to become a successful open source project with a life all of its own. Downloaded by developers from around the world on average close to a thousand times a day, it is one of the most popular JDBC drivers, commercial or open source.

0471269239_01.qxd 1/23/03 9:00 AM Page 8



Monty and David of MySQL AB eventually became aware of the size of the Java developer community wanting to use MySQL, and extended an offer for me to join their team. In June 2002, I did just that, and MM.MySQL became the official JDBC driver for MySQL. It was subsequently renamed Connector/J.

What's Next

Now you understand the need for using a database in many of the applications written today. In this chapter, we explained why MySQL is a logical choice. Using the Connector/J JDBC driver, all sorts of Java applications can access a database and its data. In the next chapter, we provide a comprehensive overview of the JDBC specification and how it has been implemented in the Connector/J driver.