CHAPTER 1

Introduction to Berkeley DB

There exist only a few pieces of software that are so versatile and so well engineered that they can be used with ease in applications as diverse as a simple web browser and a highly sophisticated telecom switch connecting millions of calls every hour. Berkeley DB is one of such pieces of software. What started as a research project at the University of California, Berkeley (UC Berkeley), has become a widely deployed and highly successful database management package, with more than 200 million known installations. And this is not the end of it—Berkeley DB has barely scratched the surface in terms of exploiting its full potential. Berkeley DB is often the most suitable database for the new breed of applications. Like search engines, peer-to-peer networking, cellular telephone and radio-frequency identification (RFID) systems, and mesh and sensor networks. What makes Berkeley DB such a success? I'll explore the answer to this question in this first chapter. I'll begin by reviewing its creation, then I'll survey its architecture and unique features.

A Brief History

Until the early 1990s, the UNIX operating system (OS) wasn't freely available to researchers and users. Anyone interested in using or studying UNIX had to get a license from AT&T. It also wasn't possible to modify the OS and make the changes available to others because AT&T owned the copyrights to the source code. In 1991, the Computer Systems Research Group at UC Berkeley decided to produce a version of UNIX that didn't require a source-code license from AT&T. It needed cheap labor to produce clean versions of key utilities that were a part of UNIX. Keith Bostic, who was leading the research group at that time, convinced Margo Seltzer and Mike Olson to work on a replacement for a UNIX utility called *ndbm*. ndbm is a UNIX utility that allows applications to manage data within the process memory. It's used in a wide variety of UNIX applications. They decided to name this utility *Berkeley DB*.

Seltzer and Bostic perfected the original 1.0 version over several releases and finally released version 1.85 in 1992. This version was rolled into Berkeley Software Distribution (BSD) version 4.4 (4.4BSD), which proved to be an instant success. Many projects, both commercial and academic, picked up the Berkeley DB code and incorporated it into applications and servers. In particular, the Lightweight Directory Access Protocol (LDAP) team at the University of Michigan chose Berkeley DB as the backing store for its directory server. By 1996, Berkeley DB became widely adopted as an in-memory database.

The University of Michigan LDAP team went on to work for an Internet startup called Netscape. It wanted to use Berkeley DB in Netscape's directory server but realized that Berkeley DB lacked a number of key features that were essential for building a commercial product on top

of it. Netscape convinced Seltzer and Bostic to add transactions, recovery, and support for concurrency to the academic version of Berkeley DB. Seltzer and Bostic incorporated Sleepycat Software, as the owner of their work, to enhance and market Berkeley DB. Sleepycat adopted *dual licensing*, a hybrid open source and proprietary strategy, to encourage the use of Berkeley DB and create a sustainable business model. Under Sleepycat, Berkeley DB was extensively improved and enhanced; a number of new products and features were added to the Berkeley DB ecosystem. The new products that were introduced include High Availability (2001), Berkeley DB XML (2003), and Berkeley DB Java Edition (2004).

By 2006, Berkeley DB had truly become a mainstream product, as companies such as Google, Yahoo!, Microsoft, Cisco, and MySQL adopted it. Eager to extend its reach in the open source world, Oracle acquired Berkeley DB in 2006.

What Is Berkeley DB?

Now that you know how Berkeley DB came into existence, let's take a look at what it actually is and what it does. Berkeley DB is a library for transactional data management. This description is simple, but it's deceptive. It doesn't give a sense of how powerful and useful it can be. Because of the ubiquity of relational databases all around us in the form of stand-alone database servers, we often associate record management with monstrous databases running on huge servers and managed by professional database administrators. It's hard for us to think of data management as something simple and closely related to the application. Part of the reason for such a perception is the success of the established relational vendors who extol the benefits of relational databases for all possible data-management problems. However, different applications need different services, and using a client/server relational engine isn't always the best way for an application to manage the data it uses.

To better understand Berkeley DB, let's take a look at the most popular types of databases.

Relational Databases

In a relational database, information is stored as a set of related tables. Each table consists of a collection of a particular type of record. You can access data in any random fashion from a relational database without having to reorganize the database tables. The biggest advantage with this approach is that a relational database can produce a different *view* of the data depending on the query executed on the database. The database is queried and modified through a generic query language called Structured Query Language (SQL). SQL statements are parsed, evaluated, and executed dynamically. The drawback with this approach is that query parsing and execution can impose a significant processing overhead.

The name *relational* comes from a branch of mathematics called *set theory*. Relations, or sets of data, can be combined or transformed in certain well-defined ways. The *relational model* is the formal definition that describes those sets and the operations that may be performed on them.

Hierarchical Databases

Hierarchical databases store information in a tree structure. Information that is closely related is stored under the same parent node in the tree. All the parent nodes that are closely related share the same grand-parent node, and so on. The advantage here is that you can retrieve

information easily and efficiently in certain fixed patterns. It's easy to find the children of any parent, so applications that need to search trees in that way get exactly the data representation that they need. However, it's difficult to find data in the tree if you need to search by attributes other than the parent-child relationship. Hierarchical databases are commonly used in applications such as directory servers.

Object Databases

An object database stores and retrieves objects that are mapped to the objects in an object-oriented programming language such as C++ or Java. Since there is a very tight coupling between the objects used in the program and the objects stored in the database, there is little query parsing and execution overhead. Object databases usually make it easy to find an object by its name (possibly a unique object ID, an address in some virtual object space, or some other unique identifier). Object databases recognize relationships—usually pointers—among objects and make them easy to traverse and search. These links can create graphs more complex than the simple parent-child hierarchies supported by a hierarchical database.

Berkeley DB

So which category does Berkeley DB fall under? The answer: none. Berkeley DB falls into a category that's a level below these higher-level abstractions. In fact, Berkeley DB can be, and has been, used to implement all of the previously mentioned databases. MySQL, the popular relational database, can be configured to use Berkeley DB as its storage engine. OpenLDAP, which is a hierarchical database, uses Berkeley DB to store data internally. Many Common Object Request Broker Architecture (CORBA) implementations use Berkeley DB to do object caching and object replication.

Berkeley DB is essentially a transactional database engine. A *transaction* ensures that the data doesn't get corrupted as it changes from one state to another. Another way to think of transactions is in terms of the so-called ACID properties. I quote the definition of the ACID properties from the venerable *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter (Morgan Kaufmann, 1993):

- **Atomicity**: A transaction's changes to state are atomic: either all happen or none happen.
- **Consistency**: A transaction is a correct transformation of the state. The actions taken as a group don't violate any of the integrity constraints associated with the state.
- **Isolation**: Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.
- **Durability**: Once a transaction completes successfully (commits), its changes to the state survive failures.

The *type* of a database—relational, hierarchical, object-oriented, and others—is independent of whether the database supports transactional data access. In fact, these types are just different ways of presenting and querying the data that they store, in a way that best suits the needs of the application.

Berkeley DB is a transactional database engine. With Berkeley DB, the application designer chooses the best access infrastructure around the core engine to suit the needs of the application. You can use Berkeley DB to design a relational database that runs as a stand-alone server, or an embedded database that runs within the process memory. Berkeley DB is a core transactional database engine, which is extremely configurable and can be used to design all sorts of application-specific databases.

Architecture of Berkeley DB

The design philosophy of Berkeley DB can be summed up as follows:

- Create a pure data-management system devoid of any application-specific constraints.
- Make the core engine very efficient.
- Provide access to the internals of the engine through application programming interfaces (APIs).

The designers of Berkeley DB believed that anticipating all the possible ways in which a database might be used would be impossible. They decided to focus instead on the common functionality that every database has to provide. Since functions such as threading, interprocess communication, and query processing vary greatly from platform to platform and application to application, they were excluded. Berkeley DB provides only these components:

- **Access methods**: Methods for creating, updating, and deleting entries and tables in the database
- **Memory pool**: A chunk of shared memory used to cache the data and share it among the processes using the database
- **Transactions**: Used to provide atomicity to a bunch of separate operations on the database
- **Locking**: A mechanism to provide concurrent access and isolation in the database
- **Buffer management**: A cache of recently used data shared among processes to reduce the frequency of disk input/output (I/O)
- **Logging**: A write-ahead logging implementation to support transactions

A unique feature of Berkeley DB is that all the subsystems, except for access methods and logging, can be used independently by the applications, outside of the context of Berkeley DB. I'll go over the details in later chapters. Figure 1-1 shows the relationships of these subsystems.

■ **Note** Relatively few applications actually use these subsystems independent of the database. This book doesn't discuss the independent use of these subsystems.

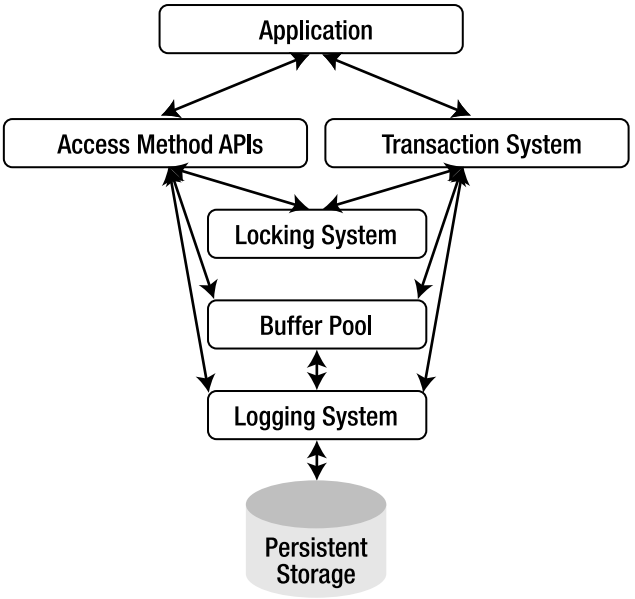

Figure 1-1. *Berkeley DB subsystems and relationships*

As shown in Figure 1-1, an application invokes the access method APIs to perform operations such as creating a database, deleting a database, inserting a record, reading a record, and so on. The application also calls the transactional subsystem to wrap multiple database operations into a single transaction.

The access methods acquire locks and then access the shared memory to retrieve the requested data and use the logging subsystem to record the operations performed. Similarly, the transactional subsystem acquires locks and accesses the shared memory region to access data and the logging module to record the operations. Finally, the buffer pool interacts with the persistent storage to service page faults and to persist data to the disk when requested.

I'll go over these subsystems in much more detail in later chapters. At this point, it's more important merely to understand the entire system at a high level than to understand the details of individual components.

Berkeley DB vs. RDBMS

Relational products are so ubiquitous and have such a huge market share that almost every conversation about database management assumes the relational model. Any database product will inevitably be compared with the relational database model. The biggest reason why the relational model is so popular and so successful is because it's extremely versatile, and exposes a single well-known standard query language—SQL—for data manipulation. The biggest problem with RDBMS is that it adds a lot of complexity and processing overhead to data management in order to support SQL. Let's look at the compromises a relational database makes to provide uniformity.

The relational model was designed to free database designers from the restriction of having to know the queries that would be run on the database in advance. The challenge is that different

applications, even when they're looking at the same data, require different types of queries. For example, a database containing the performance data for a network would be queried by the reporting application to find all the machines on which the central processing unit (CPU) usage maxed out in the last hour. A graphing application, on the other hand, might want to get all the CPU usage data gathered in the last hour. To solve this problem, the early database researchers abstracted out the query component. IBM introduced a querying language (SQL, in most of today's implementations), which could be used to express any query. This made it simple to express queries, but it introduced a significant overhead at runtime. In order to find the data a user wants, a relational system must parse a query, make a plan to find and fetch the desired records from storage, and consider ways to make the fetches efficient. Only after parsing, planning, and optimizing can the user's request actually execute. This causes the following problems:

- **Unnecessary overhead**: Applications that don't need to run different types of queries have to bear the overhead of doing query parsing and optimization, which could dominate the total query-processing time.
- **Complex query language**: To please all the vendors participating in the standards process, SQL got loaded with a large number of esoteric but rarely used features. Few SQL programmers know how to use all the language constructs correctly.
- **Unpredictable performance**: Due to a complex parsing and optimization component, it's very difficult to accurately estimate the time it takes to execute a query in a typical relational database management system (RDBMS). For real-time systems, deterministic query performance is a must.
- **Large administrative overhead**: Most relational databases are designed to handle a wide variety of data-storage needs, and usually they're deployed to take care of a number of the data-management needs of an organization. For example, a company may be managing its payroll, inventory, and production data from the same Oracle relational database instance. Administering such databases is a time-intensive and expensive proposition.
- **Large footprint**: To cater to different types of applications from a single code base, RDBMS vendors typically include a large number of features and tools with their software. Most users need only a fraction of the total feature set available. As a result, RDBMS offerings usually have a huge resource footprint (i.e., they require a lot of disk space, memory, and powerful processors to run).

Relational databases provide a good, cost-effective solution for applications that need more or less static data management, but for which the queries may vary widely over time. This combination of static data and dynamic queries is common to most relational applications. Relational applications don't perform so well when presented with dynamic data, which is mostly accessed through a fixed number of known, static queries.

Berkeley DB, in contrast, has the following features:

- **No query-processing overhead**: The application writer has to know what type of data resides in the database and must explicitly program data access according to the query patterns he expects. This is definitely more work than writing a SQL query, but this approach eliminates overhead for applications that don't need a generic querying capability.

- **No complex query language**: There's no need to learn a complex query language if all you want is a simple data store.
- **Predictable performance**: Threading, inter-process communication (IPC), and query processing introduce the most runtime overhead and performance uncertainty in a database system. Berkeley DB leaves decisions such as threading model, network boundaries, and query patterns to the application developer. As a result, the database operates using the data model and computational framework of the application. In essence, the database disappears into the application, rather than being an independent component of the deployed system.
- **Zero administrative overhead**: Berkeley DB administers itself through the database-management tasks programmed into the application by the application developer. All the management policies can be implemented using Berkeley DB API calls from inside the application.
- **Ability to be optimized for the application's access pattern**: The application developer can easily customize the database to support any access pattern.
- **Small footprint**: Berkeley DB's designers worked diligently to keep the footprint of the database extremely small. The entire package takes about 350KB of space on most platforms to install.

Berkeley DB allows you to define the access pattern most suitable for an application. Implementing the data-access layer in the application is definitely more work than writing a SQL query, but when the performance cannot be compromised, there's really no other alternative available.

Why Berkeley DB Isn't As Popular As RDBMS

After reading about all the good things about Berkeley DB, you may be asking yourself, "If Berkeley DB is so great, why isn't it as popular as RDBMS?" There are a couple of reasons behind the perception that Berkeley DB is less popular. First of all, it doesn't always get the credit it deserves. You might have used it numerous times without even noticing. For example, it's used in numerous open source applications such as Sendmail (the popular e-mail daemon) and RPM (the Linux package manager), commercial applications such as Tibco messaging products and Cisco routers, and popular online services such as Google and Amazon.com. But most of us don't know that.

And second, the relational database vendors use their huge marketing budgets and large sales teams to push any competitor out of the market. To justify the enormous investment in their complex technology, they hard-sell it even in places where it's not the best choice. That's not to say that they've succeeded only because of excessive marketing. Relational databases have proved to be extremely versatile and flexible in meeting a wide variety of data-management requirements through an easy-to-use interface. It's just that they're not suitable for applications that require a customized database solution.

Why Berkeley DB May Become More Popular

Relational databases have been tremendously successful, but that success may undermine future growth. Some developers choose to use relational systems in every application imaginable, from huge data centers to handheld organizers. There's an enormous difference between these two computing environments, and it's reasonable to look for different ways to manage data for them.

Over the past decade, the number and variety of computing devices has exploded, from servers to desktops to laptops to palmtops and beyond. At the same time, high-speed networking is becoming ubiquitous using wired and wireless technologies. The combination of distributed computing power and fast communication has created a new wave of applications. Most of these applications have internal data-management requirements and are good prospects for Berkeley DB. Some of the industry trends that drive these applications include the following:

- **Search**: Most applications and services are moving toward a search-based infrastructure, because in the new, connected world, keeping track of every resource is impossible. It's easier to search what you want when you need it than to organize everything according to a known pattern. This requires databases optimized for fast and frequent read operations.
- **Special-purpose appliances**: A number of applications that used to be run and maintained as software solutions on general-purpose hardware are being transformed into stand-alone specialized appliances. Examples include firewalls, log servers, and access points. Since applications are becoming complex and IT budgets are shrinking, companies prefer to buy solutions that don't need an administrator to install and maintain them. Appliances need self-administrating databases.
- **Miniaturization**: The size of gadgets is shrinking. Laptops are replacing desktops, and multimedia cell phones are replacing the music player, the cell phone, and the digital camera. Small footprint databases are a must for smaller devices.
- **Move toward open source**: Companies are getting fed up with the exploitative licensing schemes and expensive support contracts of the proprietary technology vendors. They now prefer to use open source technologies wherever possible.

These emerging applications are generally ill-suited to heavyweight client/server relational database management. Berkeley DB addresses the needs of these applications much better. It's only a matter of time before Berkeley DB will capture a bigger share of the database market.

Oracle Dual License

An introduction to Berkeley DB cannot be complete without an explanation of the innovative licensing scheme under which it's distributed. Berkeley DB is distributed with the a *dual license*, which states the following:

- Open source applications can freely use and redistribute Berkeley DB without paying any license fee to Sleepycat.
- Proprietary application vendors can redistribute Berkeley DB either if they make their source code, which uses Berkeley DB, public or if they buy a license from Oracle.

In essence, the dual license behaves like the GNU General Public License (GPL) for open source products and like a proprietary license for proprietary software products.

An important point to note in the case of proprietary applications is that you must buy the license only if the Berkeley DB library is *redistributed* with the application. Applications that are installed in a single location—for example, in an in-house data center—can use Berkeley DB without paying anything to Oracle and without releasing the source code of the application.

The Oracle engineering team does most of the product development and product enhancement, as it fully owns the intellectual property behind Berkeley DB. The absence of third-party components within Berkeley DB allows Oracle to sell commercial licenses without any legal ramifications. The dual license has worked very well for Berkeley DB. Adoption by the open source community has created an enormous installed base, and the licensing revenue, derived from the proprietary vendors, has allowed Oracle to keep enhancing the product.

Summary

Berkeley DB offers a set of unique features with a unique licensing scheme designed with the application designer in mind who needs a simple, effective, flexible, and reliable data-management solution. RDBMS is a versatile, general-purpose database technology that has proved its worth in a wide range of applications; however, it imposes significant processing and administrative overhead on the systems using it. Many emerging applications cannot tolerate that overhead or complexity. Berkeley DB provides an alternative for those applications, but requires more work than a relational database to integrate. In the next chapter, I'll go over the components that you'll have to build in order to use Berkeley DB.