Berkeley DB

Michael A. Olson
Keith Bostic
Margo Seltzer
Sleepycat Software, Inc.

Abstract

Berkeley DB is an Open Source embedded database system with a number of key advantages over comparable systems. It is simple to use, supports concurrent access by multiple users, and provides industrial-strength transaction support, including surviving system and disk crashes. This paper describes the design and technical features of Berkeley DB, the distribution, and its license.

1. Introduction

The Berkeley Database (Berkeley DB) is an embedded database system that can be used in applications requiring high-performance concurrent storage and retrieval of key/value pairs. The software is distributed as a library that can be linked directly into an application. It provides a variety of programmatic interfaces, including callable APIs for C, C++, Perl, Tcl and Java. Users may download Berkeley DB from Sleepycat Software's Web site, at *www.sleepycat.com*.

Sleepycat distributes Berkeley DB as an Open Source product. The company collects license fees for certain uses of the software and sells support and services.

1.1. History

Berkeley DB began as a new implementation of a hash access method to replace both `hsearch` and the various `dbm` implementations (`dbm` from AT&T, `ndbm` from Berkeley, and `gdbm` from the GNU project). In 1990 Seltzer and Yigit produced a package called Hash to do this [Selt91].

The first general release of Berkeley DB, in 1991, included some interface changes and a new B+tree access method. At roughly the same time, Seltzer and Olson developed a prototype transaction system based on Berkeley DB, called LIBTP [Selt92], but never released the code.

The 4.4BSD UNIX release included Berkeley DB 1.85 in 1992. Seltzer and Bostic maintained the code in the early 1990s in Berkeley and in Massachusetts. Many users adopted the code during this period.

By mid-1996, users wanted commercial support for the software. In response, Bostic and Seltzer formed Sleepycat Software. The company enhances, distributes, and supports Berkeley DB and supporting software and documentation. Sleepycat released version 2.1 of Berkeley DB in mid-1997 with important new features, including support for concurrent access to databases. The company makes about three commercial releases a year, and most recently shipped version 2.8.

1.2. Overview of Berkeley DB

The C interfaces in Berkeley DB permit `dbm`-style record management for databases, with significant extensions to handle duplicate data items elegantly, to deal with concurrent access, and to provide transactional support so that multiple changes can be simultaneously committed (so that they are made permanent) or rolled back (so that the database is restored to its state at the beginning of the transaction).

C++ and Java interfaces provide a small set of classes for operating on a database. The main class in both cases is called `Db`, and provides methods that encapsulate the `dbm`-style interfaces that the C interfaces provide.

Tcl and Perl interfaces allow developers working in those languages to use Berkeley DB in their applications. Bindings for both languages are included in the distribution.

Developers may compile their applications and link in Berkeley DB statically or dynamically.

1.3. How Berkeley DB is used

The Berkeley DB library supports concurrent access to databases. It can be linked into standalone applications, into a collection of cooperating applications, or into servers that handle requests and do database operations on behalf of clients.

Compared to using a standalone database management system, Berkeley DB is easy to understand and simple to use. The software stores and retrieves records, which consist of key/value pairs. Keys are used to locate items and can be any data type or structure supported by the programming language.

The programmer can provide the functions that Berkeley DB uses to operate on keys. For example, B+trees can use a custom comparison function, and the Hash access method can use a custom hash function. Berkeley DB uses default functions if none are supplied. Otherwise, Berkeley DB does not examine or interpret either keys or values in any way. Values may be arbitrarily long.

It is also important to understand what Berkeley DB is not. It is not a database server that handles network requests. It is not an SQL engine that executes queries. It is not a relational or object-oriented database management system.

It is possible to build any of those on top of Berkeley DB, but the package, as distributed, is an embedded database engine. It has been designed to be portable, small, fast, and reliable.

1.4. Applications that use Berkeley DB

Berkeley DB is embedded in a variety of proprietary and Open Source software packages. This section highlights a few of the products that use it.

Directory servers, which do data storage and retrieval using the Local Directory Access Protocol (LDAP), provide naming and directory lookup service on local-area networks. This service is, essentially, database query and update, but uses a simple protocol rather than SQL or ODBC. Berkeley DB is the embedded data manager in the majority of deployed directory servers today, including LDAP servers from Netscape, MessageDirect (formerly Isode), and others.

Berkeley DB is also embedded in a large number of mail servers. Intermail, from Software.com, uses Berkeley DB as a message store and as the backing store for its directory server. The sendmail server (including both the commercial Sendmail Pro offering from Sendmail, Inc. and the version distributed by sendmail.org) uses Berkeley DB to store aliases and other information. Similarly, Postfix (formerly VMailer) uses Berkeley DB to store administrative information.

In addition, Berkeley DB is embedded in a wide variety of other software products. Example applications include managing access control lists, storing user keys in a public-key infrastructure, recording machine-to-network-address mappings in address servers, and storing configuration and device information in video post-production software.

Finally, Berkeley DB is a part of many other Open Source software packages available on the Internet. For example, the software is embedded in the Apache Web server and the Gnome desktop.

2. Access Methods

In database terminology, an access method is the disk-based structure used to store data and the operations available on that structure. For example, many database systems support a B+tree access method. B+trees allow equality-based lookups (find keys equal to some constant), range-based lookups (find keys between two constants) and record insertion and deletion.

Berkeley DB supports three access methods: B+tree, Extended Linear Hashing (Hash), and Fixed- or Variable-length Records (Recno). All three operate on records composed of a key and a data value. In the B+tree and Hash access methods, keys can have arbitrary structure. In the Recno access method, each record is assigned a record number, which serves as the key. In all the access methods, the value can have arbitrary structure. The programmer can supply comparison or hashing functions for keys, and Berkeley DB stores and retrieves values without interpreting them.

All of the access methods use the host filesystem as a backing store.

2.1. Hash

Berkeley DB includes a Hash access method that implements extended linear hashing [Litw80]. Extended linear hashing adjusts the hash function as the hash table grows, attempting to keep all buckets underfull in the steady state.

The Hash access method supports insertion and deletion of records and lookup by exact match only. Applications may iterate over all records stored in a table, but the order in which they are returned is undefined.

2.2. B+tree

Berkeley DB includes a B+tree [Come79] access method. B+trees store records of key/value pairs in leaf pages, and pairs of (key, child page address) at internal nodes. Keys in the tree are stored in sorted order, where the order is determined by the comparison function supplied when the database was created. Pages at the leaf level of the tree include pointers to their neighbors to simplify traversal. B+trees support lookup by exact match (equality) or range (greater than or equal to a key). Like Hash tables, B+trees support record insertion, deletion, and iteration over all records in the tree.

As records are inserted and pages in the B+tree fill up, they are split, with about half the keys going into a new peer page at the same level in the tree. Most B+tree implementations leave both nodes half-full after a split. This leads to poor performance in a common case, where the caller inserts keys in order. To handle this case, Berkeley DB keeps track of the insertion order, and splits pages unevenly to keep pages fuller. This reduces tree size, yielding better search performance and smaller databases.

On deletion, empty pages are coalesced by reverse splits into single pages. The access method does no other page balancing on insertion or deletion. Keys are not moved among pages at every update to keep the tree well-balanced. While this could improve search times in some cases, the additional code complexity leads to slower updates and is prone to deadlocks.

For simplicity, Berkeley DB B+trees do no prefix compression of keys at internal or leaf nodes.

2.3. Recno

Berkeley DB includes a fixed- or variable-length record access method, called *Recno*. The Recno access method assigns logical record numbers to each record, and can search for and update records by record number. Recno is able, for example, to load a text file into a database, treating each line as a record. This permits fast searches by line number for applications like text editors [Ston82].

Recno is actually built on top of the B+tree access method and provides a simple interface for storing sequentially-ordered data values. The Recno access method generates keys internally. The programmer's view of the values is that they are numbered sequentially from one. Developers can choose to have records automatically renumbered when lower-numbered records are added or deleted. In this case, new keys can be inserted between existing keys.

3. Features

This section describes important features of Berkeley DB. In general, developers can choose which features are useful to them, and use only those that are required by their application.

For example, when an application opens a database, it can declare the degree of concurrency and recovery that it requires. Simple stand-alone applications, and in particular ports of applications that used `dbm` or one of its variants, generally do not require concurrent access or crash recovery. Other applications, such as enterprise-class database management systems that store sales transactions or other critical data, need full transactional service. Single-user operation is faster than multi-user operation, since no overhead is incurred by locking. Running with the recovery system disabled is faster than running with it enabled, since log records need not be written when changes are made to the database.

In addition, some core subsystems, including the locking system and the logging facility, can be used outside the context of the access methods as well. Although few users have chosen to do so, it is possible to use only the lock manager in Berkeley DB to control concurrency in an application, without using any of the standard database services. Alternatively, the caller can integrate locking of non-database resources with Berkeley DB's transactional two-phase locking system, to impose transaction semantics on objects outside the database.

3.1. Programmatic interfaces

Berkeley DB defines a simple API for database management. The package does not include industry-standard programmatic interfaces such as Open Database Connectivity (ODBC), Object Linking and Embedding for Databases (OleDB), or Structured Query Language (SQL). These interfaces, while useful, were designed to promote interoperability of database systems, and not simplicity or performance.

In response to customer demand, Berkeley DB 2.5 introduced support for the XA standard [Open94]. XA permits Berkeley DB to participate in distributed transactions under a transaction processing monitor like Tuxedo from BEA Systems. Like XA, other standard interfaces can be built on top of the core system. The standards do not belong inside Berkeley DB, since not all applications need them.

3.2. Working with records

A database user may need to search for particular keys in a database, or may simply want to browse available records. Berkeley DB supports both keyed access, to find one or more records with a given key, or sequential access, to retrieve all the records in the database one at a time. The order of the records returned during sequential scans depends on the access method. B+tree and Recno databases return records in sort order, and Hash databases return them in apparently random order.

Similarly, Berkeley DB defines simple interfaces for inserting, updating, and deleting records in a database.

3.3. Long keys and values

Berkeley DB manages keys and values as large as 2^{32} bytes. Since the time required to copy a record is proportional to its size, Berkeley DB includes interfaces that operate on partial records. If an application requires only part of a large record, it requests partial record retrieval, and receives just the bytes that it needs. The smaller copy saves both time and memory.

Berkeley DB allows the programmer to define the data types of keys and values. Developers use any type expressible in the programming language.

3.4. Large databases

A single database managed by Berkeley DB can be up to 2^{48} bytes, or 256 petabytes, in size. Berkeley DB uses the host filesystem as the backing store for the database, so large databases require big file support from the operating system. Sleepycat Software has customers using Berkeley DB to manage single databases in excess of 100 gigabytes.

3.5. Main memory databases

Applications that do not require persistent storage can create databases that exist only in main memory. These databases bypass the overhead imposed by the I/O system altogether.

Some applications do need to use disk as a backing store, but run on machines with very large memory. Berkeley DB is able to manage very large shared memory regions for cached data pages, log records, and lock management. For example, the cache region used for data pages may be gigabytes in size, reducing the likelihood that any read operation will need to visit the disk in the steady state. The programmer declares the size of the cache region at startup.

Finally, many operating systems provide memory-mapped file services that are much faster than their general-purpose file system interfaces. Berkeley DB can memory-map its database files for read-only database use. The application operates on records stored directly on the pages, with no cache management overhead. Because the application gets pointers directly into the Berkeley DB pages, writes cannot be permitted. Otherwise, changes could bypass the locking and logging systems, and software errors could corrupt the database. Read-only applications can use Berkeley DB's memory-mapped file service to improve performance on most architectures.

3.6. Configurable page size

Programmers declare the size of the pages used by their access methods when they create a database. Although Berkeley DB provides reasonable defaults, developers may override them to control system performance. Small pages reduce the number of records that fit on a single page. Fewer records on a page means that fewer records are locked when the page is locked, improving concurrency. The per-page overhead is proportionally higher with smaller pages, of course, but developers can trade off space for time as an application requires.

3.7. Small footprint

Berkeley DB is a compact system. The full package, including all access methods, recoverability, and transaction support is roughly 175K of text space on common architectures.

3.8. Cursors

In database terminology, a cursor is a pointer into an access method that can be called iteratively to return records in sequence. Berkeley DB includes cursor interfaces for all access methods. This permits, for example, users to traverse a B+tree and view records in order. Pointers to records in cursors are persistent, so that once fetched, a record may be updated in place. Finally, cursors support access to chains of duplicate data items in the various access methods.

3.9. Joins

In database terminology, a join is an operation that spans multiple separate tables (or in the case of Berkeley DB, multiple separate DB files). For example, a company may store information about its customers in one table and information about sales in another. An application will likely want to look up sales information by customer name; this requires matching records in the two tables that share a common customer ID field. This combining of records from multiple tables is called a join.

Berkeley DB includes interfaces for joining two or more tables.

3.10. Transactions

Transactions have four properties [Gray93]:

- They are atomic. That is, all of the changes made in a single transaction must be applied at the same instant or not at all. This permits, for example, the transfer of money between two accounts to be accomplished, by making the reduction of the balance in one account and the increase in the other into a single, atomic action.
- They must be consistent. That is, changes to the database by any transaction cannot leave the database in an illegal or corrupt state.
- They must be isolatable. Regardless of the number of users working in the database at the same time, every user must have the illusion that no other activity is going on.
- They must be durable. Even if the disk that stores the database is lost, it must be possible to recover the database to its last transaction-consistent state.

This combination of properties — atomicity, consistency, isolation, and durability — is referred to as ACIDity in the literature. Berkeley DB, like most database systems, provides ACIDity using a collection of core services.

Programmers can choose to use Berkeley DB's transaction services for applications that need them.

3.10.1. Write-ahead logging

Programmers can enable the logging system when they start up Berkeley DB. During a transaction, the application makes a series of changes to the database. Each change is captured in a log entry, which holds the state of the database record both before and after the change. The log record is guaranteed to be flushed to stable storage before any of the changed data pages are written. This behavior — writing the log before the data pages — is called *write-ahead logging*.

At any time during the transaction, the application can *commit*, making the changes permanent, or *roll back*, cancelling all changes and restoring the database to its pre-transaction state. If the application rolls back the transaction, then the log holds the state of all changed pages prior to the transaction, and Berkeley DB simply restores that state. If the application commits the transaction, Berkeley DB writes the log records to disk. In-memory copies of the data pages already reflect the changes, and will be flushed as necessary during normal processing. Since log writes are sequential, but data page writes are random, this improves performance.

3.10.2. Crashes and recovery

Berkeley DB's write-ahead log is used by the transaction system to commit or roll back transactions. It also gives the recovery system the information that it needs to protect against data loss or corruption from crashes. Berkeley DB is able to survive application crashes, system crashes, and even catastrophic failures like the loss of a hard disk, without losing any data.

Surviving crashes requires data stored in several different places. During normal processing, Berkeley DB has copies of active log records and recently-used data pages in memory. Log records are flushed to the log disk when transactions commit. Data pages trickle out to the data disk as pages move through the buffer cache. Periodically, the system administrator backs up the data disk, creating a safe copy of the database at a particular instant. When the database is backed up, the log can be truncated. For maximum robustness, the log disk and data disk should be separate devices.

Different system failures can destroy memory, the log disk, or the data disk. Berkeley DB is able to survive the loss of any one of these repositories without losing any committed transactions.

If the computer's memory is lost, through an application or operating system crash, then the log holds all committed transactions. On restart, the recovery system rolls the log forward against the database, reapplying any changes to on-disk pages that were in memory at the time of the crash. Since the log contains pre- and post-change state for transactions, the recovery system also uses the log to restore any pages to their original state if they were modified by transactions that never committed.

If the data disk is lost, the system administrator can restore the most recent copy from backup. The recovery system will roll the entire log forward against the original database, reapplying all committed changes. When it finishes, the database will contain every change made by every transaction that ever committed.

If the log disk is lost, then the recovery system can use the in-memory copies of log entries to roll back any uncommitted transactions, flush all in-memory database pages to the data disk, and shut down gracefully. At that point, the system administrator can back up the database disk, install a new log disk, and restart the system.

3.10.3. Checkpoints

Berkeley DB includes a checkpointing service that interacts with the recovery system. During normal processing, both the log and the database are changing continually. At any given instant, the on-disk versions of the two are not guaranteed to be consistent. The log probably contains changes that are not yet in the database.

When an application makes a *checkpoint*, all committed changes in the log up to that point are guaranteed to be present on the data disk, too. Checkpointing is moderately expensive during normal processing, but limits the time spent recovering from crashes.

After an application or operating system crash, the recovery system only needs to go back two checkpoints[1] to start rolling the log forward. Without checkpoints, there is no way to be sure how long restarting after a crash will take. With checkpoints, the restart interval can be fixed by the programmer. Recovery processing can be guaranteed to complete in a second or two.

Software crashes are much more common than disk failures. Many developers want to guarantee that software bugs do not destroy data, but are willing to restore from tape, and to tolerate a day or two of lost work, in the unlikley event of a disk crash. With Berkeley DB, programmers may truncate the log at checkpoints. As long as the two most recent checkpoints are present, the recovery system can guarantee that no committed transactions are lost after a software crash. In this case, the recovery system does not require that the log and the data be on separate devices, although separating them can still improve performance by spreading out writes.

3.10.4. Two-phase locking

Berkeley DB provides a service known as two-phase locking. In order to reduce the likelihood of deadlocks and to guarantee ACID properties, database systems manage locks in two phases. First, during the operation of a transaction, they acquire locks, but never release them. Second, at the end of the transaction, they release locks, but never acquire them. In practice, most database systems, including Berkeley DB, acquire locks on demand over the course of the transaction, then flush the log, then release all locks.

[1] One checkpoint is not far enough. The recovery system cannot be sure that the most recent checkpoint completed — it may have been interrupted by the crash that forced the recovery system to run in the first place.

Berkeley DB can lock entire database files, which correspond to tables, or individual pages in them. It does no record-level locking. By shrinking the page size, however, developers can guarantee that every page holds only a small number of records. This reduces contention.

If locking is enabled, then read and write operations on a database acquire two-phase locks, which are held until the transaction completes. Which objects are locked and the order of lock acquisition depend on the workload for each transaction. It is possible for two or more transactions to deadlock, so that each is waiting for a lock that is held by another.

Berkeley DB detects deadlocks and automatically rolls back one of the transactions. This releases the locks that it held and allows the other transactions to continue. The caller is notified that its transaction did not complete, and may restart it. Developers can specify the deadlock detection interval and the policy to use in choosing a transaction to roll back.

The two-phase locking interfaces are separately callable by applications that link Berkeley DB, though few users have needed to use that facility directly. Using these interfaces, Berkeley DB provides a fast, platform-portable locking system for general-purpose use. It also lets users include non-database objects in a database transaction, by controlling access to them exactly as if they were inside the database.

The Berkeley DB two-phase locking facility is built on the fastest correct locking primitives that are supported by the underlying architecture. In the current implementation, this means that the locking system is different on the various UNIX platforms, and is still more different on Windows NT. In our experience, the most difficult aspect of performance tuning is finding the fastest locking primitives that work correctly on a particular architecture and then integrating the new interface with the several that we already support.

The world would be a better place if the operating systems community would uniformly implement POSIX locking primitives and would guarantee that acquiring an uncontested lock was a fast operation. Locks must work both among threads in a single process and among processes.

3.11. Concurrency

Good performance under concurrent operation is a critical design point for Berkeley DB. Although Berkeley DB is itself not multi-threaded, it is thread-safe, and runs well in threaded applications. Philosophically, we view the use of threads and the choice of a threads package as a policy decision, and prefer to offer mechanism (the ability to run threaded or not), allowing applications to choose their own policies.

The locking, logging, and buffer pool subsystems all use shared memory or other OS-specific sharing facilities to communicate. Locks, buffer pool fetches, and log writes behave in the same way across threads in a single process as they do across different processes on a single machine.

As a result, concurrent database applications may start up a new process for every single user, may create a single server which spawns a new thread for every client request, or may choose any policy in between.

Berkeley DB has been carefully designed to minimize contention and maximize concurrency. The cache manager allows all threads or processes to benefit from I/O done by one. Shared resources must sometimes be locked for exclusive access by one thread of control. We have kept critical sections small, and are careful not to hold critical resource locks across system calls that could deschedule the locking thread or process. Sleepycat Software has customers with hundreds of concurrent users working on a single database in production.

4. Engineering Philosophy

Fundamentally, Berkeley DB is a collection of access methods with important facilities, like logging, locking, and transactional access underlying them. In both the research and the commercial world, the techniques for building systems like Berkeley DB have been well-known for a long time.

The key advantage of Berkeley DB is the careful attention that has been paid to engineering details throughout its life. We have carefully designed the system so that the core facilities, like locking and I/O, surface the right interfaces and are otherwise opaque to the caller. As programmers, we understand the value of simplicity and have worked hard to simplify the interfaces we surface to users of the database system.

Berkeley DB avoids limits in the code. It places no practical limit on the size of keys, values, or databases; they may grow to occupy the available storage space.

The locking and logging subsystems have been carefully crafted to reduce contention and improve throughput by shrinking or eliminating critical sections, and reducing the sizes of locked regions and log entries.

There is nothing in the design or implementation of Berkeley DB that pushes the state of the art in database systems. Rather, we have been very careful to get the engineering right. The result is a system that is superior, as an embedded database system, to any other solution available.

Most database systems trade off simplicity for correctness. Either the system is easy to use, or it supports concurrent use and survives system failures. Berkeley DB, because of its careful design and implementation, offers both simplicity and correctness.

The system has a small footprint, makes simple operations simple to carry out (inserting a new record takes just a few lines of code), and behaves correctly in the face of heavy concurrent use, system crashes, and even catastrophic failures like loss of a hard disk.

5. The Berkeley DB 2.x Distribution

Berkeley DB is distributed in source code form from *www.sleepycat.com*. Users are free to download and build the software, and to use it in their applications.

5.1. What is in the distribution

The distribution is a compressed archive file. It includes the source code for the Berkeley DB library, as well as documentation, test suites, and supporting utilities.

The source code includes build support for all supported platforms. On UNIX systems Berkeley DB uses the GNU autoconfiguration tool, `autoconf`, to identify the system and to build the library and supporting utilities. Berkeley DB includes specific build environments for other platforms, such as VMS and Windows.

5.1.1. Documentation

The distributed system includes documentation in HTML format. The documentation is in two parts: a UNIX-style reference manual for use by programmers, and a reference guide which is tutorial in nature.

5.1.2. Test suite

The software also includes a complete test suite, written in Tcl. We believe that the test suite is a key advantage of Berkeley DB over comparable systems.

First, the test suite allows users who download and build the software to be sure that it is operating correctly.

Second, the test suite allows us, like other commercial developers of database software, to exercise the system thoroughly at every release. When we learn of new bugs, we add them to the test suite. We run the test suite continually during development cycles, and always prior to release. The result is a much more reliable system by the time it reaches beta release.

5.2. Binary distribution

Sleepycat makes compiled libraries and general binary distributions available to customers for a fee.

5.3. Supported platforms

Berkeley DB runs on any operating system with a POSIX 1003.1 interface [IEEE96], which includes virtually every UNIX system. In addition, the software runs on VMS, Windows/95, Windows/98, and Windows/NT. Sleepycat Software no longer supports deployment on sixteen-bit Windows systems.

6. Berkeley DB 2.x Licensing

Berkeley DB 2.x is distributed as an Open Source product. The software is freely available from us at our Web site, and in other media. Users are free to download the software and build applications with it.

The 1.x versions of Berkeley DB were covered by the UC Berkeley copyright that covers software freely redistributable in source form. When Sleepycat Software was formed, we needed to draft a license consistent with the copyright governing the existing, older software. Because of important differences between the UC Berkeley copyright and the GPL, it was impossible for us to use the GPL. A second copyright, with terms contradictory to the first, simply would not have worked.

Sleepycat wanted to continue Open Source development of Berkeley DB for several reasons. We agree with Raymond [Raym98] and others that Open Source software is typically of higher quality than proprietary, binary-only products. Our customers benefit from a community of developers who know and use Berkeley DB, and can help with application design, debugging, and performance tuning. Widespread distribution and use of the source code tends to isolate bugs early, and to get fixes back into the distributed system quickly. As a result, Berkeley DB is more reliable. Just as importantly, individual users are able to contribute new features and performance enhancements, to the benefit of everyone who uses Berkeley DB. From a business perspective, Open Source and free distribution of the software creates share for us, and gives us a market into which we can sell products and services. Finally, making the source code freely available reduces our support load, since customers can find and fix bugs without recourse to us, in many cases.

To preserve the Open Source heritage of the older Berkeley DB code, we drafted a new license governing the distribution of Berkeley DB 2.x. We adopted terms from the GPL that make it impossible to turn our Open Source code into proprietary code owned by someone else.

Briefly, the terms governing the use and distribution of Berkeley DB are:

- your application must be internal to your site, or
- your application must be freely redistributable in source form, or
- you must get a license from us.

For customers who prefer not to distribute Open Source products, we sell licenses to use and extend Berkeley DB at a reasonable cost.

We work hard to accommodate the needs of the Open Source community. For example, we have crafted special licensing arrangements with Gnome to encourage its use and distribution of Berkeley DB.

Berkeley DB conforms to the Open Source definition [Open99]. The license has been carefully crafted to keep the product available as an Open Source offering, while providing enough of a return on our investment to fund continued development and support of the product. The current license has created a business capable of funding three years of development on the software that simply would not have happened otherwise.

7. Summary

Berkeley DB offers a unique collection of features, targeted squarely at software developers who need simple, reliable database management services in their applications. Good design and implementation and careful engineering throughout make the software better than many other systems.

Berkeley DB is an Open Source product, available at *www.sleepycat.com* for download. The distributed system includes everything needed to build and deploy the software or to port it to new systems.

Sleepycat Software distributes Berkeley DB under a license agreement that draws on both the UC Berkeley copyright and the GPL. The license guarantees that Berkeley DB will remain an Open Source product and provides Sleepycat with opportunities to make money to fund continued development on the software.

8. References

[Come79]

Comer, D., “The Ubiquitous B-tree,” *ACM Computing Surveys* Volume 11, number 2, June 1979.

[Gray93]

Gray, J., and Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan-Kaufman Publishers, 1993.

[IEEE96]

Institute for Electrical and Electronics Engineers, *IEEE/ANSI Std 1003.1*, 1996 Edition.

[Litw80]

Litwin, W., “Linear Hashing: A New Tool for File and Table Addressing,” *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, Montreal, Quebec, Canada, October 1980.

[Open94]

The Open Group, *Distributed TP: The XA+ Specification, Version 2*, The Open Group, 1994.

[Open99]

Opensource.org, “Open Source Definition,” *www.opensource.org/osd.html*, version 1.4, 1999.

[Raym98]

Raymond, E.S., “The Cathedral and the Bazaar,” *www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html*, January 1998.

[Selt91]

Seltzer, M., and Yigit, O., “A New Hashing Package for UNIX,” *Proceedings 1991 Winter USENIX Conference*, Dallas, TX, January 1991.

[Selt92]

Seltzer, M., and Olson, M., “LIBTP: Portable Modular Transactions for UNIX,” *Proceedings 1992 Winter Usenix Conference*, San Francisco, CA, January 1992.

[Ston82]

Stonebraker, M., Stettner, H., Kalash, J., Guttman, A., and Lynn, N., “Document Processing in a Relational Database System,” Memorandum No. UCB/ERL M82/32, University of California at Berkeley, Berkeley, CA, May 1982.