Berkeley DB: A Retrospective

Margo Seltzer, Oracle Corporation

Abstract

Berkeley DB is an open-source, embedded transactional data management system that has been in wide deployment since 1992. In those fifteen years, it has grown from a simple, non-transactional key-data store to a highly reliable, scalable, flexible data management solution. We trace the history of Berkeley DB and discuss how a small library provides data management solutions appropriate to disparate environments ranging form cell phones to servers.

1 Introduction

Berkeley DB is an open-source, embedded transactional data management system that has been in wide deployment since 1992. In this context, embedded means that it is a library that is linked with an application, providing data management completely hidden from the end-user. Unlike conventional database management solutions, Berkeley DB does not have a separate server process and requires no manual database administration. All administration is built directly into the applications that embed it.

As Berkeley DB is an embedded data management system, claims of performance and scale are only meaningful in the context of applications that use it. Berkeley DB provides data management for mission-critical web applications and web sites. For example, Google Accounts uses Berkeley DB to store all user and service account information and preferences [12]. If this application/database goes down, users cannot access their email and other online services; reliability is crucial. Distributed and replicated storage provides the reliability guarantees that Google requires. Similarly, Amazon's front page user-customization makes, on average, 800 requests to a Berkeley DB database. An unavailable database loses customers.

Berkeley DB scales in terms of the amount of data it manages, the capabilities of the devices on which it runs, and the distance over which applications distribute data. Berkeley DB databases range from bytes to terabytes, with our largest installations reaching petabytes. Similarly, Berkeley DB runs on everything from small, handheld devices like Motorola cell phones to wireless routers [15] to large, multiprocessor servers. Berkeley DB's replication and high availability features are used both across backplanes and across the Internet.

The rest of this paper is organized as follows. In Section 2, we present the history of Berkeley DB. In section 3, we describe the Berkeley DB product family as it exists today. In section 4, we discuss how Berkeley DB addresses the widely varying needs of the environments in which it is embedded and the challenges this flexibility introduces. We conclude in Section 5.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 A Brief History of DB

Berkeley DB began life in 1991 as a dynamic linear hashing implementation [13] designed to replace the historic dbm [1], ndbm [5], and hsearch [2] APIs. Prior to its release as a library in the 4.4 BSD release, Keith Bostic of the UC Berkeley Computer Science Research Group (CSRG) designed an access-method independent API and Mike Olson, also at UC Berkeley, added a Btree implementation under that API. In 1992, UC Berkeley released 4.4BSD including the hash and Btree implementations backing the generic ”db” interface that has become known as db-1.85.

In 1992, Olson and Seltzer published a paper describing a package called LIBTP [14]. LIBTP was a transactional implementation of db-1.85 that demonstrated impressive performance, relative to conventional client-server database systems. The implementation was a research prototype that was never released.

Db-1.85 enjoyed widespread adoption after its release with 4.4BSD. In particular, both the MIT Kerberos project and the University of Michigan LDAP project incorporated it. A group at Harvard released a minor patch version db-1.86 for the Kerberos team in 1996. In November of 1996, Seltzer and Bostic started Sleepycat Software to develop and market a transactional implementation of Berkeley DB. The transactional implementation was released in 1997 as Berkeley DB 2.0, the first commercial release of Berkeley DB.

The Sleepycat releases of Berkeley DB were made available under the Sleepycat Public License, which is now known as a dual license. The code is open source and may be downloaded and used freely. However, redistribution requires that either the package using Berkeley DB be released as open source or that the distributors obtain a commercial license from Sleepycat Software.

In 1999, Sleepycat Software released Berkeley DB 3.0, which transformed the original APIs into an object-oriented handle and method style interface. In 2001, Sleepycat Software released Berkeley DB 4.0, which included single-master, multiple-reader replication. This product, Berkeley DB/High Availability, offers outstanding availability, because replicas can take over for a failed master, and outstanding scalability, because read-only replicas can reduce master load. In 2006, Oracle acquired Sleepycat Software to round out its embedded database offerings. The Sleepycat products continue to be developed, maintained and supported by Oracle and are still available under a dual license.

3 Berkeley DB Today

The Berkeley DB product family today includes the original Berkeley DB library as well as an XML product (Berkeley DB XML) that layers an XML schema and Xquery and Xpath capabilities atop the original Berkeley DB library. Berkeley DB Java Edition (JE) provides functionality comparable to Berkeley DB in a 100% pure Java implementation that has been designed to work well in today’s Java Virtual Machines. Figure 1 shows the Berkeley DB product family architecture.

The rest of this paper will focus on the Berkeley DB C-library. Other papers [6, 9] describe Berkeley DB XML and Berkeley DB Java Edition.

3.1 Overview

We begin this section with a brief overview of Berkeley DB, omitting many of the details that appear in other places [10]. We then discuss Berkeley DB’s replication and high availability in greater detail. Section 4 then goes on to discuss the flexibility applications enjoy when they use Berkeley DB and the challenges introduced by this flexibility.

Berkeley DB provides high-performance persistent storage of key data pairs. Both keys and data are opaque byte-strings; Berkeley DB databases have no schema. The application that embeds Berkeley DB is responsible for imposing its own schema on the data. The advantage of this approach is that an application is free to store data in whatever form is most natural to it. If the application manipulates objects, the objects can be stored

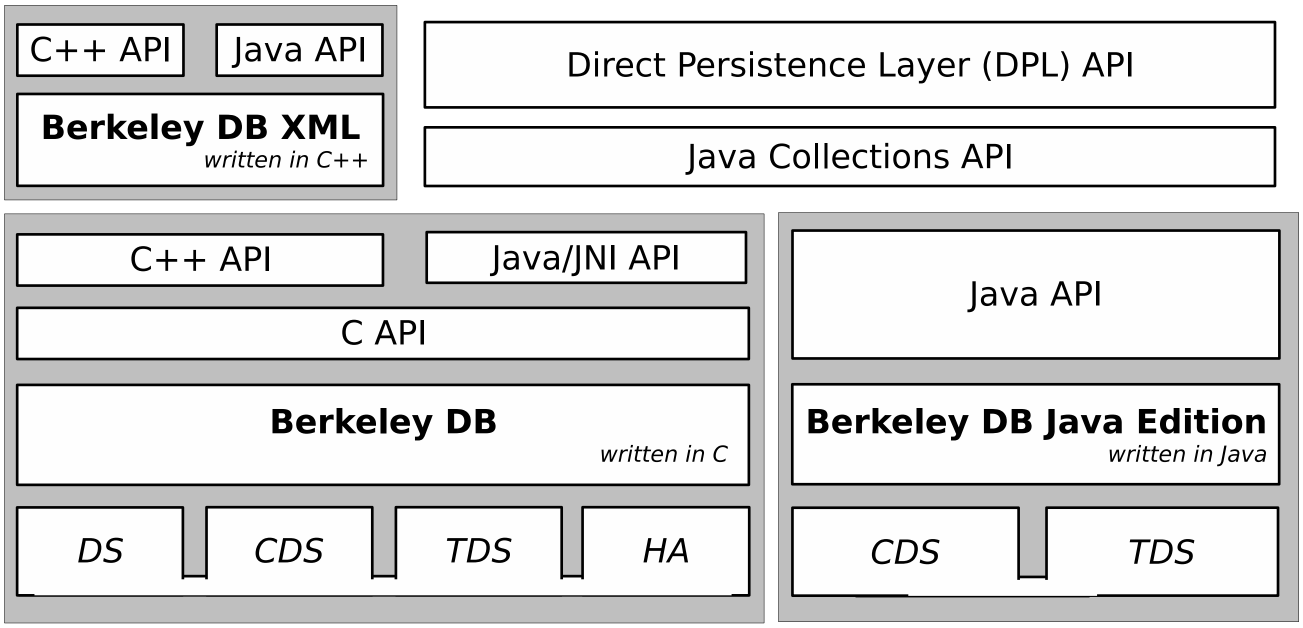

Figure 1: Berkeley DB Product Family: Berkeley DB consists of three products. The original transactional C-library key/value store is shown in the lower left corner, including APIs in Java and C++. Berkeley DB XML occupies the upper left corner, where it sits atop Berkeley DB. Berkeley DB Java Edition (JE) is in the lower right. Note that the Java Collections API and Direct Persistence Layer are APIs that sit atop either JE or the Java/JNI interface of the C-library.

directly in Berkeley DB; if the application manipulates relational rows, those too can be stored. In fact, different data formats can be stored in the same databases, as long as the application understands how to interpret the data items. This provides maximum application flexibility.

Berkeley DB provides several different data structures for indexing key values. The Btree index is by far the most widely used index type, but the library provides hash and queue indexes and a record-number-based index implemented on top of the Btree index. Applications *put* key/value pairs into databases, retrieve them via *get* calls, and remove them using *delete* methods. Indexes can require that all key values be unique or they can be allowed to accept duplicate values for the same key. All such policy decisions are left up to the application.

Applications access key/value pairs through handles on databases (somewhat akin to relational tables) or through *cursor* handles. Cursors are primarily used for iteration, representing a specific place within a database. Databases are implemented on top of the native file system; a file may contain one or more databases.

Transactional APIs provide the ability to wrap one or more operations inside a transaction, providing the ability to abort a sequence of operations and also providing recovery from application, operating system, or hardware failure. Unlike many other systems, database-level operations, such as database create, rename, and remove, can also be encapsulated in transactions.

3.2 Replication provides Scalability and High Availability

Following the basic design philosophy of Berkeley DB, the replication features provide mechanisms without specifying policies. The library provides the ability to create and manage any number of replicas, but lets the application control the degree of replication, how tightly synchronized the replicas are, how the replicas communicate, etc.

Berkeley DB replication is a log-shipping system. A replication group consists of a single *master* and one or more read-only *replicas*. All write operations, such as database create, key/value insert, update, or delete, and database destroy, must be processed transactionally by the master. The master sends log records to each of the replicas. The replicas apply log records only when they receive a transaction commit record. Therefore, aborted transactions introduce almost no load on the replicas; replicas will record log records from aborted transactions, but will not perform any work on their behalf.

If the master fails, one of the replicas can take over as master. In most cases, applications use the Berkeley DB paxos-compliant [8] election mechanism to select the new master, however, applications can choose to select a new master themselves. This latter approach is most frequently used when the application is running on top of a high availability substrate that includes master selection logic.

The Berkeley DB election algorithm is designed to prevent transaction loss. The replicas all send votes to each other indicating their application-specified priority, their maximum log record, and a unique tie-breaker. Each replica tallies the votes, selecting the winning site first based upon log records, then upon priority, and finally upon tie-breaker value. Thus, a low priority site with a complete set of log records will be elected before a higher priority site that is missing log records. After an election, each replica synchronizes with the master and then begins accepting and applying log records from the new master as it had done before the election.

Berkeley DB replication provides both scalability and high availability. It achieves scalability, because an application can deploy as many read-only replicas as necessary to scale with the read load. It delivers high availability, because sub-second elections with automatic failover and recovery provide the appearance of non-stop operation. Berkeley DB High Availability has been in widespread deployment for over five years in mission critical services, such as Google's Single Sign On service [12].

4 Flexibility Opportunities and Challenges

Configuration flexibility is critical for Berkeley DB, because it embeds in such a wide range of environments. The demands of a cell phone environment are dramatically different from those of a server farm running thousands of instances of a Berkeley DB application. Berkeley DB is successful in this huge range of environments by providing applications enormous flexibility in how they use the database. In this section, we discuss some of the most commonly used configuration settings; for more detailed discussion of configuration options, please see the Berkeley DB documentation [11].

There are three different ways to take advantage of Berkeley DB flexibility. First, Berkeley DB provides a range of build-time options that drive compilation of the library. Second, there are four different feature sets from which application developers can select; each feature set introduces trade-offs in functionality and performance. Third, there are run-time options that control the behavior of the library.

4.1 Compile Time Options

Compile-time configuration is provided by Berkeley DB's autoconfiguration [7] script. The two most common compile-time configuration options provide a small footprint build (*–enable-smallbuild*) and activate higher concurrency locking (*–enable-fine-grained-lock-manager*). These two options are designed for radically different environments and illustrate clearly the value of configurability.

The smallbuild configuration is designed for resource-constrained environments that need the power of the Berkeley DB library while consuming as few resources as possible. In fact, this configuration option arose from the constraints of a cell phone environment. When configured with smallbuild, the library contains only the Btree index (hash, queue, and record number are all omitted). It also omits replication, cryptography, statistics collection, and verification. The resulting library is approximately one-half megabyte and contains everything necessary to use transactional, recoverable btrees.

The fine-grained lock manager option is designed for large, highly concurrenct applications – think large servers running in a data center with hundreds or thousands of concurrent threads. Transactional applications must acquire locks to protect data while it is being read or written. These locks are maintained in shared memory data structures, whose integrity is maintained by acquiring and releasing mutexes while the data structures are being manipulated. The conventional Berkeley DB lock manager uses a single mutex to protect the entire lock shared memory region. Mutexes quickly become performance bottlenecks as they serialize access to memory. The advantage of this approach is that the application requires only a single mutex acquire/release pair for any lock operations. However, the disadvantage is that the mutex may be held for many thousands of instructions, limiting the degree of parallelism in the application, if it is lock intensive.

The fine-grained lock manager option trades off the number of mutex acquisition/release pairs against the advantage of having multiple threads or processes accessing shared data structures simultaneously. When this option is enabled, an application will acquire a larger number of mutexes, however, it will hold these mutexes for less time. Therefore, an application will observe reduced single-threaded performance, but better scalability as more threads of control are active simultaneously. This option should be considered carefully and used only when it provides significant benefit.

4.2 Feature Set Selection

The Berkeley DB library has four distinct feature sets as shown in Figure 1. The Data Store (DS) feature set is most similar to the original Berkeley DB 1.85 library. It provides high-performance storage for key/data pairs, but does not provide locking, transactions, or replication. The obvious benefit of this feature set is that it introduces the least overhead – no locking, no logging, no synchronous writes due to transaction commit. However, the disadvantages are also obvious – there can be no concurrent access by readers and writers and there is no recovery from any kind of failure. This approach is usually most suitable for temporary data storage.

The Concurrent Data Store (CDS) feature set addresses the lack of concurrency, but not recovery from failure. In CDS, the library locks at the API-layer, enforcing the constraint that there be either a single writer or multiple readers in the library, relieving the application of the burden of managing its own concurrency. CDS is most beneficial in the presence of concurrent workloads that are read-dominated with infrequent updates. It incurs slightly more overhead than DS, because it does acquire locks, but it incurs less overhead than the transactional feature set, because the library acquires only a single lock per API invocation, rather than a lock per page as is done with the transactional product.

Transactional Data Store (TDS) is currently the most widely used feature set. It provides both highly concurrent access and recovery from application, operating system, and hardware failure. The advantages to this configuration are numerous: multiple threads of control can access and modify the database simultaneously, data is never lost even after a failure, and the application can commit and abort transactions as necessary. However, this functionality comes at some performance cost. The library must log all updates, thereby increasing I/O. The library also obtains locks on database pages to prevent multiple threads from modifying the same page at the same time. Finally, the library will (normally) issue synchronous I/Os to commit transactions. Nonetheless, the benefits of the transactional feature set typically far outweigh these costs.

The High Availability (HA) feature set introduces the most complicated set of trade-offs. It provides the recoverability of TDS with the additional benefit that an application can continue running even after a site fails. It may seem that such functionality must incur a more significant performance penalty than TDS, but that is not always the case. In many cases, because many copies of the data exist, applications choose to commit transactions to remote replicas rather than require synchronous disk I/O. Commiting to the network is often faster than commiting to disk, so HA can offer *better* performance than TDS.

4.3 Runtime Configuration

Berkeley DB provides the greatest latitude at runtime where applications can make reliability/performance trade-offs and select settings most appropriate for a particular environment. There are far more runtime configuration options that can be discussed here, so we focus on those that applications most frequently use. First we talk about index structure selection and tuning, then discuss trade-offs between durability and performance, and conclude with a discussion of different concurrency management techniques.

As discussed above, Berkeley DB provides several different indexing data structures. In addition to selecting the appropriate index type, applications can select the underlying page size as well. Small pages improve concurrency, since Berkeley DB performs page-based locking; large pages typically improve I/O performance. The library will select a reasonable default page size that is well-matched to the underlying file system, but sometimes applications possess more information and can achieve better performance by explicitly setting a page size.

Another way that applications can tune indexes is by specifying application-specific key sorting and/or hash functions. By default, Berkeley DB uses lexicographic sorting on ordered keys and a simple XOR-based hash function. However, when applications are intimately familiar with the domain of their key values, an application can frequently select a sort or hash function more appropriate to the data.

The second area where applications frequently make critical configuration decisions is in trading off durability and performance. Berkeley DB must issue a synchronous disk I/O to achieve transactional durability on commit. The I/O is typically sequential, because it is a log write, but it still requires leaving high performance solid state memory and accessing a slow, mechanical device. Applications that can sacrifice durability guarantees can request that Berkeley DB commit asynchronously. In this case, the library will write a commit log record, but will not force it to disk. This means that upon failure, the library will still preserve transaction boundaries and leave the data in a transactionally consistent state, but some of the last transactions to commit may be lost. A slightly stronger form of this technique is also available; the library will write the log records to the operating system's buffer cache, but will not force the disk write. This avoids nearly all of the cost of the log write and protects against transaction loss when the application crashes, but the system continues to run. As discussed above, avoiding the disk write at commit is frequently used when an application is already replicating its data to other sites.

Berkeley DB allows applications to take disk-write avoidance to the extreme: applications can run completely in memory. In a pure in-memory environment, all databases, log files, etc are stored in memory. Obviously, if the application crashes, all data is lost, however, in some environments sacrificing durability for higher performance is attractive. For example, consider a router. While the router is running, performance is critical, and even if some components in the router fail, as long as some components are running, availability is crucial (i.e., you do not want to drop connections). However, if the router crashes, restoring connection state information after a restart provides no value. In such a configuration, Berkeley DB's pure in-memory configuration is attractive. Data and log records are never written to disk, but both are replicated across multiple hardware instances. In this manner, the system provides both outstanding availability and outstanding performance.

Concurrency control techniques are another mechanism that can dramatically affect performance and scalability. By default, Berkeley DB uses conventional two-phase locking (2PL) [4]. In 2PL, the library acquires a lock of the appropriate mode (read or write) before accessing or modifying data, holding the locks until commit. The two phases of 2PL refer to a first phase during which the library acquires locks and a second phase during which it releases locks. In Berkeley DB, as in most systems, the first phase lasts during the entire duration of the transaction and the second phase is fully encapsulated at commit or abort time.

Two-phase locking usually works quite well, but can limit concurrency during database traversal, because a database traversal will ultimately lock the entire database. When such traversals are common, multiversion concurrency control (MVCC) [3] provides significant benefits.

In MVCC, writers make copies of database pages rather than acquiring locks on them. While such writing transactions are active, readers can continue to access the original copies of the pages. Thus, database traversal can proceed even in the presence of update transactions. MVCC provides increased concurrency for some workloads at the expense of increased memory consumption and data copies. Berkeley DB provides both two-phase locking and multi-version concurrency control.

4.4 Challenges

Berkelely DB's tremendous flexibility introduces a set of unique challenges. Features that are obviously desirable for one environment may be completely unacceptable for another. For example, some customers wonder why Berkeley DB does not assume TCP/IP as its replication communication protocol. While TCP/IP makes sense in some environments, it is not acceptable in an environment where Berkeley DB is replicated across a backplane. Historically, Berkeley DB has left any and all such policy decisions to the application. Unfortunately, that placed a heavy burden on the application developer. An alternative approach is to offer such solutions, allowing developers to override them when necessary. This philosophy is embodied in the recently-introduced *Replication Framework*.

The Replication Framework makes several policy decisions regarding how replication is managed. For example, TCP/IP is the communication infrastructure and threading is provided via pthreads. The advantage is that application developers write significantly less code. The disadvantage is that these decisions limit some of the library's flexibility. This is a tension that will always exist in Berkeley DB as long as it continues to be an appropriate solution from cell phones to servers.

The fact that Berkeley DB is a library that links into an application introduces a unique set of problems. The library design must be correct regardless of the application developer's choices for surrounding components. For example, the library must work correctly regardless of the choice of thread package, file system, mutex implementation, etc.

Adding open-source to the library equation introduces additional issues. Because Berkeley DB is available in source-code form, it becomes an obvious choice for developers working on new or unusual hardware platforms. Thus, portability is a requirement. When customers are compiling your software, they will use any compiler available, so the software must be portable across all compilers. Although languages have been ”standardized” for years, each compiler introduces its own idiosyncracies and constraints. For example, a recent release of glibc made Berkeley DB fail to compile, because the system includes contain a macro definition for open, and multiple Berkeley DB data structures contain an open method.

Code quality also takes on a significant role when customers can (and do) read your source code. A poorly expressed comment can jeopardize sales when potential customers read the code and question the comments. Even comments that allude to not-yet-implemented features can be problematic. Nothing beats open source for encouraging rigorous adherence to coding standards and clear, correct commenting.

Another constant tension concerns what features are best provided by the library and what features are better provided by the embedding application. As discussed by Perl and Seltzer [12], *master leases* provide a guaranteed ability to perform reads that never return stale data. They conclude that while Google was able to implement master leases in the application, it would have been better if such functionality were available in the library. In a similar vein, the Berkeley DB replication API does not provide a mechanism to manage replication group membership; this capability resides in the replication framework. Nonetheless, it would be useful functionality to applications that cannot be constrained by other choices in the replication framework.

In general, Berkeley DB incorporates new features when there is clear demonstrated customer demand from multiple market segments. Such decision making requires flexibility in both the library, the Berkeley DB designers, and those application developers using Berkeley DB.

5 Conclusions

Berkeley DB is a dramatically different library than its db-1.85 ancestor. Its roots are clearly visible, but it has matured gracefully over time. What began life as a simple key/value storage engine now provides mission critical data storage for many key Internet infrastructure services.

6 Bibliography

References

[1] AT&T, DBM(3X), *UNIX Programmer's Manual, Seventh Edition*, Volume 1, January", 1979.

[2] AT&T, HSEARCH(BA-LIB), *UNIX System User's 's Manual, System V*, Volume 3:506–508, 1985.

[3] P. Bernstein and N. Goodman, "Concurrency control algorithms for multiversion database systems", *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 209–215, ACM Press, Ottawa, Canada, 1982.

[4] P. Bernstein and V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987.

[5] Computer Systems Research Group, NDBM(3), *4.3 BSD UNIX Programmer's Manual Reference Guide*, University of California, Berkeley", 1986.

[6] P. Ford "Berkeley DB XML: An Embedded XML Database", *XML.com*, O'Reilly and Associates, May, 2003.

[7] GNU. autoconf. http://www.gnu.org/software/autoconf.

[8] L. Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[9] J. Menard, "Building Applications with Berkeley DB Java Edition", *Java Developer's Journal*, Sys-Con Media, September 2004.

[10] M. Olson and K. Bostic and M. Seltzer, "Berkeley DB", *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, Monterey, CA, June 1999.

[11] Oracle Corporation. Berkeley DB Documentation. http://www.oracle.com/technology/documentation/berkeley-db/db/index.html.

[12] S. Perl and M. Seltzer, "Data Management for Internet-Scale Single-Sign-On", *Proceedings of the 3rd Workshop on Real, Large Distributed Systems (WORLDS '06)*, Seattle, WA, November 2006.

[13] M. Seltzer and O. Yigit, "A New Hashing Package for UNIX", *Proceedings of the 1991 Winter USENIX Conference*, 173–184, USENIX Association, Dallas, TX, 1991.

[14] M. Seltzer and M. Olson "LIBTP: Portable, Modular Transactions for UNIX", *Proceedings of the 1992 Winter USENIX Conference*, 9–26, "USENIX Association, San Francisco, CA, 1992.

[15] R. Van Tassle and M. Richardson. Wireless networking & Berkeley DB. *Dr. Dobbs Journal*, 27(1), CMP Media, January 2002.