# An introduction to data management

Cheap, powerful computing and networking have created countless new applications that could not have existed a decade ago. The advent of the World-Wide Web, and its influence in driving the Internet into homes and businesses, is one obvious example. Equally important, though, is the from large, general-purpose desktop and server computers toward smaller, special-purpose devices with built-in processing and communications services.

As computer hardware has spread into virtually every corner of our lives, of course, software has followed. Software developers today are building applications not just for conventional desktop and server environments, but also for handheld computers, home appliances, networking hardware, cars and trucks, factory floor automation systems, and more.

While these operating environments are diverse, the problems that software engineers must solve in them are often strikingly similar. Most systems must deal with the outside world, whether that means communicating with users or controlling machinery. As a result, most need some sort of I/O system. Even a simple, single-function system generally needs to handle multiple tasks, and so needs some kind of operating system to schedule and manage control threads. Also, many computer systems must store and retrieve data to track history, record configuration settings, or manage access.

Data management can be very simple. In some cases, just recording configuration in a flat text file is enough. More often, though, programs need to store and search a large amount of data, or structurally complex data. Database management systems are tools that programmers can use to do this work quickly and efficiently using off-the-shelf software.

Of course, database management systems have been around for a long time. Data storage is a problem dating back to the earliest days of computing. Software developers can choose from hundreds of good, commercially-available database systems. The problem is selecting the one that best solves the problems that their applications face.

**Mapping the terrain: theory and practice**

The first step in selecting a database system is figuring out what the choices are. Decades of research and real-world deployment have produced countless systems. We need to organize them somehow to reduce the number of options.

One obvious way to group systems is to use the common labels that vendors apply to them. The buzzwords here include "network," "relational," "object-oriented," and "embedded," with some cross-fertilization like "object-relational" and "embedded network". Understanding the buzzwords is important. Each has some grounding in theory, but has also evolved into a practical label for categorizing systems that work in a certain way.

All database systems, regardless of the buzzwords that apply to them, provide a few common services. All of them store data, for example. We'll begin by exploring the common services that all systems provide, and then examine the differences among the different kinds of systems.

**Data access and data management**

Fundamentally, database systems provide two services.

The first service is *data access*. Data access means adding new data to the database (inserting), finding data of interest (searching), changing data already stored (updating), and removing data from the database (deleting). All databases provide these services. How they work varies from category to category, and depends on the record structure that the database supports.

Each record in a database is a collection of values. For example, the record for a Web site customer might include a name, email address, shipping address, and payment information. Records are usually stored in tables. Each table holds records of the same kind. For example, the **customer** table at an e-commerce Web site might store the customer records for every person who shopped at the site. Often, database records have a different structure from the structures or instances supported by the programming language in which an application is written. As a result, working with records can mean:

* using database operations like searches and updates on records; and
* converting between programming language structures and database record types in the application.

The second service is *data management*. Data management is more complicated than data access. Providing good data management services is the hard part of building a database system. When you choose a database system to use in an application you build, making sure it supports the data management services you need is critical.

Data management services include allowing multiple users to work on the database simultaneously (concurrency), allowing multiple records to be changed instantaneously (transactions), and surviving application and system crashes (recovery). Different database systems offer different data management services. Data management services are entirely independent of the data access services listed above. For example, nothing about relational database theory requires that the system support transactions, but most commercial relational systems do.

Concurrency means that multiple users can operate on the database at the same time. Support for concurrency ranges from none (single-user access only) to complete (many readers and writers working simultaneously).

Transactions permit users to make multiple changes appear at once. For example, a transfer of funds between bank accounts needs to be a transaction because the balance in one account is reduced and the balance in the other increases. If the reduction happened before the increase, than a poorly-timed system crash could leave the customer poorer; if the bank used the opposite order, then the same system crash could make the customer richer. Obviously, both the customer and the bank are best served if both operations happen at the same instant.

Transactions have well-defined properties in database systems. They are *atomic*, so that the changes happen all at once or not at all. They are *consistent*, so that the database is in a legal state when the transaction begins and when it ends. They are typically *isolated*, which means that any other users in the database cannot interfere with them while they are in progress. And they are *durable*, so that if the system or application crashes after a transaction finishes, the changes are not lost. Together, the properties of *atomicity*, *consistency*, *isolation*, and *durability* are known as the ACID properties.

As is the case for concurrency, support for transactions varies among databases. Some offer atomicity without making guarantees about durability. Some ignore isolatability, especially in single-user systems; there's no need to isolate other users from the effects of changes when there are no other users.

Another important data management service is recovery. Strictly speaking, recovery is a procedure that the system carries out when it starts up. The purpose of recovery is to guarantee that the database is complete and usable. This is most important after a system or application crash, when the database may have been damaged. The recovery process guarantees that the internal structure of the database is good. Recovery usually means that any completed transactions are checked, and any lost changes are reapplied to the database. At the end of the recovery process, applications can use the database as if there had been no interruption in service.

Finally, there are a number of data management services that permit copying of data. For example, most database systems are able to import data from other sources, and to export it for use elsewhere. Also, most systems provide some way to back up databases and to restore in the event of a system failure that damages the database. Many commercial systems allow *hot backups*, so that users can back up databases while they are in use. Many applications must run without interruption, and cannot be shut down for backups.

A particular database system may provide other data management services. Some provide browsers that show database structure and contents. Some include tools that enforce data integrity rules, such as the rule that no employee can have a negative salary. These data management services are not common to all systems, however. Concurrency, recovery, and transactions are the data management services that most database vendors support.

Deciding what kind of database to use means understanding the data access and data management services that your application needs. Berkeley DB is an embedded database that supports fairly simple data access with a rich set of data management services. To highlight its strengths and weaknesses, we can compare it to other database system categories.

**Relational databases**

Relational databases are probably the best-known database variant, because of the success of companies like Oracle. Relational databases are based on the mathematical field of set theory. The term "relation" is really just a synonym for "set" -- a relation is just a set of records or, in our terminology, a table. One of the main innovations in early relational systems was to insulate the programmer from the physical organization of the database. Rather than walking through arrays of records or traversing pointers, programmers make statements about tables in a high-level language, and the system executes those statements.

Relational databases operate on *tuples*, or records, composed of values of several different data types, including integers, character strings, and others. Operations include searching for records whose values satisfy some criteria, updating records, and so on.

Virtually all relational databases use the Structured Query Language, or SQL. This language permits people and computer programs to work with the database by writing simple statements. The database engine reads those statements and determines how to satisfy them on the tables in the database.

SQL is the main practical advantage of relational database systems. Rather than writing a computer program to find records of interest, the relational system user can just type a query in a simple syntax, and let the engine do the work. This gives users enormous flexibility; they do not need to decide in advance what kind of searches they want to do, and they do not need expensive programmers to find the data they need. Learning SQL requires some effort, but it's much simpler than a full-blown high-level programming language for most purposes. And there are a lot of programmers who have already learned SQL.

**Object-oriented databases**

Object-oriented databases are less common than relational systems, but are still fairly widespread. Most object-oriented databases were originally conceived as persistent storage systems closely wedded to particular high-level programming languages like C++. With the spread of Java, most now support more than one programming language, but object-oriented database systems fundamentally provide the same class and method abstractions as do object-oriented programming languages.

Many object-oriented systems allow applications to operate on objects uniformly, whether they are in memory or on disk. These systems create the illusion that all objects are in memory all the time. The advantage to object-oriented programmers who simply want object storage and retrieval is clear. They need never be aware of whether an object is in memory or not. The application simply uses objects, and the database system moves them between disk and memory transparently. All of the operations on an object, and all its behavior, are determined by the programming language.

Object-oriented databases aren't nearly as widely deployed as relational systems. In order to attract developers who understand relational systems, many of the object-oriented systems have added support for query languages very much like SQL. In practice, though, object-oriented databases are mostly used for persistent storage of objects in C++ and Java programs.

**Network databases**

The "network model" is a fairly old technique for managing and navigating application data. Network databases are designed to make pointer traversal very fast. Every record stored in a network database is allowed to contain pointers to other records. These pointers are generally physical addresses, so fetching the record to which it refers just means reading it from disk by its disk address.

Network database systems generally permit records to contain integers, floating point numbers, and character strings, as well as references to other records. An application can search for records of interest. After retrieving a record, the application can fetch any record to which it refers, quickly.

Pointer traversal is fast because most network systems use physical disk addresses as pointers. When the application wants to fetch a record, the database system uses the address to fetch exactly the right string of bytes from the disk. This requires only a single disk access in all cases. Other systems, by contrast, often must do more than one disk read to find a particular record.

The key advantage of the network model is also its main drawback. The fact that pointer traversal is so fast means that applications that do it will run well. On the other hand, storing pointers all over the database makes it very hard to reorganize the database. In effect, once you store a pointer to a record, it is difficult to move that record elsewhere. Some network databases handle this by leaving forwarding pointers behind, but this defeats the speed advantage of doing a single disk access in the first place. Other network databases find, and fix, all the pointers to a record when it moves, but this makes reorganization very expensive. Reorganization is often necessary in databases, since adding and deleting records over time will consume space that cannot be reclaimed without reorganizing. Without periodic reorganization to compact network databases, they can end up with a considerable amount of wasted space.

**Clients and servers**

Database vendors have two choices for system architecture. They can build a server to which remote clients connect, and do all the database management inside the server. Alternatively, they can provide a module that links directly into the application, and does all database management locally. In either case, the application developer needs some way of communicating with the database (generally, an Application Programming Interface (API) that does work in the process or that communicates with a server to get work done).

Almost all commercial database products are implemented as servers, and applications connect to them as clients. Servers have several features that make them attractive.

First, because all of the data is managed by a separate process, and possibly on a separate machine, it's easy to isolate the database server from bugs and crashes in the application.

Second, because some database products (particularly relational engines) are quite large, splitting them off as separate server processes keeps applications small, which uses less disk space and memory. Relational engines include code to parse SQL statements, to analyze them and produce plans for execution, to optimize the plans, and to execute them.

Finally, by storing all the data in one place and managing it with a single server, it's easier for organizations to back up, protect, and set policies on their databases. The enterprise databases for large companies often have several full-time administrators caring for them, making certain that applications run quickly, granting and denying access to users, and making backups.

However, centralized administration can be a disadvantage in some cases. In particular, if a programmer wants to build an application that uses a database for storage of important information, then shipping and supporting the application is much harder. The end user needs to install and administer a separate database server, and the programmer must support not just one product, but two. Adding a server process to the application creates new opportunity for installation mistakes and run-time problems.

**What is Berkeley DB?**

So far, we've discussed database systems in general terms. It's time now to consider Berkeley DB in particular and see how it fits into the framework we have introduced. The key question is, what kinds of applications should use Berkeley DB?

Berkeley DB is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications. Berkeley DB provides a simple function-call API for data access and management.

By "open source," we mean that Berkeley DB is distributed under a license that conforms to the [Open Source Definition](http://www.opensource.org/osd.html). This license guarantees that Berkeley DB is freely available for use and redistribution in other open source products. [Sleepycat Software](http://www.sleepycat.com/) sells commercial licenses for redistribution in proprietary applications, but in all cases the complete source code for Berkeley DB is freely available for download and use.

Berkeley DB is embedded because it links directly into the application. It runs in the same address space as the application. As a result, no inter-process communication, either over the network or between processes on the same machine, is required for database operations. Berkeley DB provides a simple function-call API for a number of programming languages, including C, C++, Java, Perl, Tcl, Python, and PHP. All database operations happen inside the library. Multiple processes, or multiple threads in a single process, can all use the database at the same time as each uses the Berkeley DB library. Low-level services like locking, transaction logging, shared buffer management, memory management, and so on are all handled transparently by the library.

The library is extremely portable. It runs under almost all UNIX and Linux variants, Windows, and a number of embedded real-time operating systems. It runs on both 32-bit and 64-bit systems. It has been deployed on high-end Internet servers, desktop machines, and on palmtop computers, set-top boxes, in network switches, and elsewhere. Once Berkeley DB is linked into the application, the end user generally does not know that there's a database present at all.

Berkeley DB is scalable in a number of respects. The database library itself is quite compact (under 300 kilobytes of text space on common architectures), but it can manage databases up to 256 terabytes in size. It also supports high concurrency, with thousands of users operating on the same database at the same time. Berkeley DB is small enough to run in tightly constrained embedded systems, but can take advantage of gigabytes of memory and terabytes of disk on high-end server machines.

Berkeley DB generally outperforms relational and object-oriented database systems in embedded applications for a couple of reasons. First, because the library runs in the same address space, no inter-process communication is required for database operations. The cost of communicating between processes on a single machine, or among machines on a network, is much higher than the cost of making a function call. Second, because Berkeley DB uses a simple function-call interface for all operations, there is no query language to parse, and no execution plan to produce.

**Data Access Services**

Berkeley DB applications can choose the storage structure that best suits the application. Berkeley DB supports hash tables, Btrees, simple record-number-based storage, and persistent queues. Programmers can create tables using any of these storage structures, and can mix operations on different kinds of tables in a single application.

Hash tables are generally good for very large databases that need predictable search and update times for random-access records. Hash tables allow users to ask, "Does this key exist?" or to fetch a record with a known key. Hash tables do not allow users to ask for records with keys that are close to a known key.

Btrees are better for range-based searches, as when the application needs to find all records with keys between some starting and ending value. Btrees also do a better job of exploiting *locality of reference*. If the application is likely to touch keys near each other at the same time, the Btrees work well. The tree structure keeps keys that are close together near one another in storage, so fetching nearby values usually doesn't require a disk access.

Record-number-based storage is natural for applications that need to store and fetch records, but that do not have a simple way to generate keys of their own. In a record number table, the record number is the key for the record. Berkeley DB will generate these record numbers automatically.

Queues are well-suited for applications that create records, and then must deal with those records in creation order. A good example is on-line purchasing systems. Orders can enter the system at any time, but should generally be filled in the order in which they were placed.

**Data management services**

Berkeley DB offers important data management services, including concurrency, transactions, and recovery. All of these services work on all of the storage structures.

Many users can work on the same database concurrently. Berkeley DB handles locking transparently, ensuring that two users working on the same record do not interfere with one another.

The library provides strict ACID transaction semantics, by default. However, applications are allowed to relax the isolation guarantees the database system makes.

Multiple operations can be grouped into a single transaction, and can be committed or rolled back atomically. Berkeley DB uses a technique called *two-phase locking* to be sure that concurrent transactions are isolated from one another, and a technique called *write-ahead logging* to guarantee that committed changes survive application, system, or hardware failures.

When an application starts up, it can ask Berkeley DB to run recovery. Recovery restores the database to a clean state, with all committed changes present, even after a crash. The database is guaranteed to be consistent and all committed changes are guaranteed to be present when recovery completes.

An application can specify, when it starts up, which data management services it will use. Some applications need fast, single-user, non-transactional Btree data storage. In that case, the application can disable the locking and transaction systems, and will not incur the overhead of locking or logging. If an application needs to support multiple concurrent users, but doesn't need transactions, it can turn on locking without transactions. Applications that need concurrent, transaction-protected database access can enable all of the subsystems.

In all these cases, the application uses the same function-call API to fetch and update records.

**Design**

Berkeley DB was designed to provide industrial-strength database services to application developers, without requiring them to become database experts. It is a classic C-library style *toolkit*, providing a broad base of functionality to application writers. Berkeley DB was designed by programmers, for programmers: its modular design surfaces simple, orthogonal interfaces to core services, and it provides mechanism (for example, good thread support) without imposing policy (for example, the use of threads is not required). Just as importantly, Berkeley DB allows developers to balance performance against the need for crash recovery and concurrent use. An application can use the storage structure that provides the fastest access to its data and can request only the degree of logging and locking that it needs.

Because of the tool-based approach and separate interfaces for each Berkeley DB subsystem, you can support a complete transaction environment for other system operations. Berkeley DB even allows you to wrap transactions around the standard UNIX file read and write operations! Further, Berkeley DB was designed to interact correctly with the native system's toolset, a feature no other database package offers. For example, Berkeley DB supports hot backups (database backups while the database is in use), using standard UNIX system utilities, for example, dump, tar, cpio, pax or even cp.

Finally, because scripting language interfaces are available for Berkeley DB (notably Tcl and Perl), application writers can build incredibly powerful database engines with little effort. You can build transaction-protected database applications using your favorite scripting languages, an increasingly important feature in a world using CGI scripts to deliver HTML.

# What Berkeley DB is not

In contrast to most other database systems, Berkeley DB provides relatively simple data access services.

Records in Berkeley DB are (*key*, *value*) pairs. Berkeley DB supports only a few logical operations on records. They are:

* Insert a record in a table.
* Delete a record from a table.
* Find a record in a table by looking up its key.
* Update a record that has already been found.

Notice that Berkeley DB never operates on the value part of a record. Values are simply payload, to be stored with keys and reliably delivered back to the application on demand.

Both keys and values can be arbitrary byte strings, either fixed-length or variable-length. As a result, programmers can put native programming language data structures into the database without converting them to a foreign record format first. Storage and retrieval are very simple, but the application needs to know what the structure of a key and a value is in advance. It cannot ask Berkeley DB, because Berkeley DB doesn't know.

This is an important feature of Berkeley DB, and one worth considering more carefully. On the one hand, Berkeley DB cannot provide the programmer with any information on the contents or structure of the values that it stores. The application must understand the keys and values that it uses. On the other hand, there is literally no limit to the data types that can be store in a Berkeley DB database. The application never needs to convert its own program data into the data types that Berkeley DB supports. Berkeley DB is able to operate on any data type the application uses, no matter how complex.

Because both keys and values can be up to four gigabytes in length, a single record can store images, audio streams, or other large data values. Large values are not treated specially in Berkeley DB. They are simply broken into page-sized chunks, and reassembled on demand when the application needs them. Unlike some other database systems, Berkeley DB offers no special support for binary large objects (BLOBs).

### Not a relational database

Berkeley DB is not a relational database.

First, Berkeley DB does not support SQL queries. All access to data is through the Berkeley DB API. Developers must learn a new set of interfaces in order to work with Berkeley DB. Although the interfaces are fairly simple, they are non-standard.

SQL support is a double-edged sword. One big advantage of relational databases is that they allow users to write simple declarative queries in a high-level language. The database system knows everything about the data and can carry out the command. This means that it's simple to search for data in new ways, and to ask new questions of the database. No programming is required.

On the other hand, if a programmer can predict in advance how an application will access data, then writing a low-level program to get and store records can be faster. It eliminates the overhead of query parsing, optimization, and execution. The programmer must understand the data representation, and must write the code to do the work, but once that's done, the application can be very fast.

Second, Berkeley DB has no notion of *schema* and data types in the way that relational systems do. Schema is the structure of records in tables, and the relationships among the tables in the database. For example, in a relational system the programmer can create a record from a fixed menu of data types. Because the record types are declared to the system, the relational engine can reach inside records and examine individual values in them. In addition, programmers can use SQL to declare relationships among tables, and to create indices on tables. Relational engines usually maintain these relationships and indices automatically.

In Berkeley DB, the key and value in a record are opaque to Berkeley DB. They may have a rich internal structure, but the library is unaware of it. As a result, Berkeley DB cannot decompose the value part of a record into its constituent parts, and cannot use those parts to find values of interest. Only the application, which knows the data structure, can do that. Berkeley DB does support indices on tables and automatically maintain those indices as their associated tables are modified.

Berkeley DB is not a relational system. Relational database systems are semantically rich and offer high-level database access. Compared to such systems, Berkeley DB is a high-performance, transactional library for record storage. It's possible to build a relational system on top of Berkeley DB. In fact, the popular MySQL relational system uses Berkeley DB for transaction-protected table management, and takes care of all the SQL parsing and execution. It uses Berkeley DB for the storage level, and provides the semantics and access tools.

### Not an object-oriented database

Object-oriented databases are designed for very tight integration with object-oriented programming languages. Berkeley DB is written entirely in the C programming language. It includes language bindings for C++, Java, and other languages, but the library has no information about the objects created in any object-oriented application. Berkeley DB never makes method calls on any application object. It has no idea what methods are defined on user objects, and cannot see the public or private members of any instance. The key and value part of all records are opaque to Berkeley DB.

Berkeley DB cannot automatically page in objects as they are accessed, as some object-oriented databases do. The object-oriented application programmer must decide what records are required, and must fetch them by making method calls on Berkeley DB objects.

### Not a network database

Berkeley DB does not support network-style navigation among records, as network databases do. Records in a Berkeley DB table may move around over time, as new records are added to the table and old ones are deleted. Berkeley DB is able to do fast searches for records based on keys, but there is no way to create a persistent physical pointer to a record. Applications can only refer to records by key, not by address.

### Not a database server

Berkeley DB is not a standalone database server. It is a library, and runs in the address space of the application that uses it. If more than one application links in Berkeley DB, then all can use the same database at the same time; the library handles coordination among the applications, and guarantees that they do not interfere with one another.

Recent releases of Berkeley DB allow programmers to compile the library as a standalone process, and to use RPC stubs to connect to it and to carry out operations. However, there are some important limitations to this feature. The RPC stubs provide exactly the same API that the library itself does. There is no higher-level access provided by the standalone process. Tuning the standalone process is difficult, since Berkeley DB does no threading in the library (applications can be threaded, but the library never creates a thread on its own).

It is possible to build a server application that uses Berkeley DB for data management. For example, many commercial and open source Lightweight Directory Access Protocol (LDAP) servers use Berkeley DB for record storage. LDAP clients connect to these servers over the network. Individual servers make calls through the Berkeley DB API to find records and return them to clients. On its own, however, Berkeley DB is not a server.

# Do you need Berkeley DB?

Berkeley DB is an ideal database system for applications that need fast, scalable, and reliable embedded database management. For applications that need different services, however, it can be a poor choice.

First, do you need the ability to access your data in ways you cannot predict in advance? If your users want to be able to enter SQL queries to perform complicated searches that you cannot program into your application to begin with, then you should consider a relational engine instead. Berkeley DB requires a programmer to write code in order to run a new kind of query.

On the other hand, if you can predict your data access patterns up front -- and in particular if you need fairly simple key/value lookups -- then Berkeley DB is a good choice. The queries can be coded up once, and will then run very quickly because there is no SQL to parse and execute.

Second, are there political arguments for or against a standalone relational server? If you're building an application for your own use and have a relational system installed with administrative support already, it may be simpler to use that than to build and learn Berkeley DB. On the other hand, if you'll be shipping many copies of your application to customers, and don't want your customers to have to buy, install, and manage a separate database system, then Berkeley DB may be a better choice.

Third, are there any technical advantages to an embedded database? If you're building an application that will run unattended for long periods of time, or for end users who are not sophisticated administrators, then a separate server process may be too big a burden. It will require separate installation and management, and if it creates new ways for the application to fail, or new complexities to master in the field, then Berkeley DB may be a better choice.

The fundamental question is, how closely do your requirements match the Berkeley DB design? Berkeley DB was conceived and built to provide fast, reliable, transaction-protected record storage. The library itself was never intended to provide interactive query support, graphical reporting tools, or similar services that some other database systems provide. We have tried always to err on the side of minimalism and simplicity. By keeping the library small and simple, we create fewer opportunities for bugs to creep in, and we guarantee that the database system stays fast, because there is very little code to execute. If your application needs that set of features, then Berkeley DB is almost certainly the best choice for you.

# What other services does Berkeley DB provide?

Berkeley DB also provides core database services to developers. These services include:

Page cache management:

The page cache provides fast access to a cache of database pages, handling the I/O associated with the cache to ensure that dirty pages are written back to the file system and that new pages are allocated on demand. Applications may use the Berkeley DB shared memory buffer manager to serve their own files and pages.

Transactions and logging:

The transaction and logging systems provide recoverability and atomicity for multiple database operations. The transaction system uses two-phase locking and write-ahead logging protocols to ensure that database operations may be undone or redone in the case of application or system failure. Applications may use Berkeley DB transaction and logging subsystems to protect their own data structures and operations from application or system failure.

Locking:

The locking system provides multiple reader or single writer access to objects. The Berkeley DB access methods use the locking system to acquire the right to read or write database pages. Applications may use the Berkeley DB locking subsystem to support their own locking needs.

By combining the page cache, transaction, locking, and logging systems, Berkeley DB provides the same services found in much larger, more complex and more expensive database systems. Berkeley DB supports multiple simultaneous readers and writers and guarantees that all changes are recoverable, even in the case of a catastrophic hardware failure during a database update.

Developers may select some or all of the core database services for any access method or database. Therefore, it is possible to choose the appropriate storage structure and the right degrees of concurrency and recoverability for any application. In addition, some of the subsystems (for example, the Locking subsystem) can be called separately from the Berkeley DB access method. As a result, developers can integrate non-database objects into their transactional applications using Berkeley DB.

# What does the Berkeley DB distribution include?

The Berkeley DB distribution includes complete source code for the Berkeley DB library, including all three Berkeley DB products and their supporting utilities, as well as complete documentation in HTML format.

The distribution does not include prebuilt binaries or libraries, or hard-copy documentation. Prebuilt libraries and binaries for some architecture/compiler combinations are available as part of Sleepycat Software's Berkeley DB support services.

# Where does Berkeley DB run?

Berkeley DB requires only underlying IEEE/ANSI Std 1003.1 (POSIX) system calls and can be ported easily to new architectures by adding stub routines to connect the native system interfaces to the Berkeley DB POSIX-style system calls. See [Porting Berkeley DB to new architectures](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/distrib/port.html) for more information.

Berkeley DB will autoconfigure and run on almost any modern UNIX, POSIX or Linux systems, and on most historical UNIX platforms. Berkeley DB will autoconfigure and run on almost any GNU gcc toolchain-based embedded platform, including Cygwin, Embedix, OpenLinux and others. See [Building for UNIX systems](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/build_unix/intro.html) for more information.

The Berkeley DB distribution includes support for QNX Neutrino. See [Building for UNIX systems](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/build_unix/intro.html) for more information.

The Berkeley DB distribution includes support for VxWorks, via a workspace and project files for Tornado 2.0. See [Building for VxWorks](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/build_vxworks/intro.html) for more information.

The Berkeley DB distribution includes support for Windows/95, Windows/98, Windows/NT, Windows/2000 and Windows/XP, via the Microsoft Visual C++ 6.0 and .NET development environments. See[Building for Windows systems](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/build_win/intro.html) for more information.

# Sleepycat Software's Berkeley DB products

Sleepycat Software licenses four different products that use the Berkeley DB technology. Each product offers a distinct level of database support. It is not possible to mix-and-match products, that is, each application or group of applications must use the same Berkeley DB product.

All four products are included in the single Open Source distribution of Berkeley DB from Sleepycat Software, and building that distribution automatically builds all four products. Each product adds new interfaces and services to the product that precedes it in the list. As a result, developers can download Berkeley DB and build an application that does only single-user, read-only database access, and easily add support later for more users and more complex database access patterns.

Users who distribute Berkeley DB must ensure that they are licensed for the Berkeley DB interfaces they use. Information on licensing is available directly from Sleepycat Software.

### Berkeley DB Data Store

The Berkeley DB Data Store product is an embeddable, high-performance data store. It supports multiple concurrent threads of control to read information managed by Berkeley DB. When updates are required, only a single process may be using the database. That process may be multithreaded, but only one thread of control should be allowed to update the database at any time. The Berkeley DB Data Store does no locking, and so provides no guarantees of correct behavior if more than one thread of control is updating the database at a time. The Berkeley DB Data Store is intended for use in single-user or read-only applications that can guarantee that no more than one thread of control will ever update the database at any time.

### Berkeley DB Concurrent Data Store

The Berkeley DB Concurrent Data Store product adds multiple-reader, single writer capabilities to the Berkeley DB Data Store product, supporting applications that need concurrent updates and do not want to implement their own locking protocols. Berkeley DB Concurrent Data Store is intended for applications that require occasional write access to a database that is largely used for reading.

### Berkeley DB Transactional Data Store

The Berkeley DB Transactional Data Store product adds full transactional support and recoverability to the Berkeley DB Data Store product. Berkeley DB Transactional Data Store is intended for applications that require industrial-strength database services, including excellent performance under high-concurrency workloads with a mixture of readers and writers, the ability to commit or roll back multiple changes to the database at a single instant, and the guarantee that even in the event of a catastrophic system or hardware failure, any committed database changes will be preserved.

### Berkeley DB High Availability

The Berkeley DB High Availability product support for data replication. A single master system handles all updates, and distributes them to as many replicas as the application requires. All replicas can handle read requests during normal processing. If the master system fails for any reason, one of the replicas takes over as the new master system, and distributes updates to the remaining replicas.

1. **A Simple Access Method Tutorial**

**Introduction**

As an introduction to Berkeley DB, we will present a few Berkeley DB programming concepts, and then a simple database application.

The programming concepts are:

* [Key/data pairs](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/keydata.html)
* [Object handles](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/handles.html)
* [Error returns](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/errors.html)

This database application will:

* [Create a simple database](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/open.html)
* [Store items](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/put.html)
* [Retrieve items](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/get.html)
* [Remove items](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/del.html)
* [Close the database](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/close.html)

The introduction will be presented using the programming language C. The [complete source](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/simple_tut/example.cs) of the final version of the example program is included in the Berkeley DB distribution.

# Key/data pairs

Berkeley DB uses key/data pairs to identify elements in the database. That is, in the general case, whenever you call a Berkeley DB interface, you present a key to identify the key/data pair on which you intend to operate.

For example, you might store some key/data pairs as follows:

|  |  |
| --- | --- |
| **Key:** | **Data:** |
| fruit | apple |
| sport | cricket |
| drink | water |

In each case, the first element of the pair is the key, and the second is the data. To store the first of these key/data pairs into the database, you would call the Berkeley DB interface to store items, with **fruit** as the key, and **apple** as the data. At some future time, you could then retrieve the data item associated with **fruit**, and the Berkeley DB retrieval interface would return **apple** to you. While there are many variations and some subtleties, all accesses to data in Berkeley DB come down to key/data pairs.

Both key and data items are stored in simple structures (called [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html)s) that contain a reference to memory and a length, counted in bytes. (The name [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) is an acronym for *database thang*, chosen because nobody could think of a sensible name that wasn't already in use somewhere else.) Key and data items can be arbitrary binary data of practically any length, including 0 bytes. There is a single data item for each key item, by default, but databases can be configured to support multiple data items for each key item.

# Object handles

With a few minor exceptions, Berkeley DB functionality is accessed by creating a structure and then calling functions that are fields in that structure. This is, of course, similar to object-oriented concepts, of instances and methods on them. For simplicity, we will often refer to these structure fields as methods of the handle.

The manual pages will show these methods as C structure references. For example, the open-a-database method for a database handle is represented as [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html)

# Error returns

The Berkeley DB interfaces always return a value of 0 on success. If the operation does not succeed for any reason, the return value will be non-zero.

If a system error occurred (for example, Berkeley DB ran out of disk space, or permission to access a file was denied, or an illegal argument was specified to one of the interfaces), Berkeley DB returns an **errno**value. All of the possible values of **errno** are greater than 0.

If the operation didn't fail due to a system error, but wasn't successful either, Berkeley DB returns a special error value. For example, if you tried to retrieve the data item associated with the key **fruit**, and there was no such key/data pair in the database, Berkeley DB would return [DB\_NOTFOUND](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/program/errorret.html#DB_NOTFOUND), a special error value that means the requested key does not appear in the database. All of the possible special error values are less than 0.

Berkeley DB also offers programmatic support for displaying error return values. First, the [db\_strerror](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/env_strerror.html) interface returns a pointer to the error message corresponding to any Berkeley DB error return, similar to the ANSI C strerror interface, but is able to handle both system error returns and Berkeley DB-specific return values.

Second, there are two error functions, [DB->err](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_err.html) and [DB->errx](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_err.html). These functions work like the ANSI C printf interface, taking a printf-style format string and argument list, and optionally appending the standard error string to a message constructed from the format string and other arguments.

# Opening a database

Opening a database is done in two steps: first, a [DB](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_class.html) handle is created using the Berkeley DB [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html) interface, and then the actual database is opened using the [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) function.

The [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html) interface takes three arguments:

dbp

A location to store a reference to the created structure.

environment

A location to specify an enclosing Berkeley DB environment, not used in our example.

flags

A placeholder for flags, not used in our example.

The [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) interface takes five arguments:

file

The name of the database file to be opened.

database

The optional database name, not used in this example.

type

The type of database to open. This value will be one of the four access methods Berkeley DB supports: DB\_BTREE, DB\_HASH, DB\_QUEUE or DB\_RECNO, or the special value DB\_UNKNOWN, which allows you to open an existing file without knowing its type.

flags

Various flags that modify the behavior of [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html). In our simple case, the only interesting flag is [DB\_CREATE](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/env_open.html#DB_CREATE). This flag behaves similarly to the IEEE/ANSI Std 1003.1 (POSIX) O\_CREATE flag to the open system call, causing Berkeley DB to create the underlying database if it does not yet exist.

mode

The file mode of any underlying files that [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) will create. The mode behaves as does the IEEE/ANSI Std 1003.1 (POSIX) mode argument to the open system call, and specifies file read, write and execute permissions. Of course, only the read and write permissions are relevant to Berkeley DB.

Here's what the code to create the handle and then call [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) looks like:

**#include <sys/types.h>**

**#include <stdio.h>**

**#include <db.h>**

**#define DATABASE "access.db"**

**int**

**main()**

**{**

**DB \*dbp;**

**int ret;**

**if ((ret = db\_create(&dbp, NULL, 0)) != 0) {**

**fprintf(stderr, "db\_create: %s\n", db\_strerror(ret));**

**exit (1);**

**}**

**if ((ret = dbp->open(dbp,**

**NULL, DATABASE, NULL, DB\_BTREE, DB\_CREATE, 0664)) != 0) {**

**dbp->err(dbp, ret, "%s", DATABASE);**

**goto err;**

**}**

If the call to [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html) is successful, the variable **dbp** will contain a database handle that will be used to configure and access an underlying database.

As you see, the program opens a database named **access.db**. The underlying database is a Btree. Because the [DB\_CREATE](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/env_open.html#DB_CREATE) flag was specified, the file will be created if it does not already exist. The mode of any created files will be 0664 (that is, readable and writable by the owner and the group, and readable by everyone else).

One additional function call is used in this code sample, [DB->err](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_err.html). This method works like the ANSI C printf interface. The second argument is the error return from a Berkeley DB function, and the rest of the arguments are a printf-style format string and argument list. The error message associated with the error return will be appended to a message constructed from the format string and other arguments. In the above code, if the [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) call were to fail, the message it would display would be something like

access.db: Operation not permitted

# Adding elements to a database

The simplest way to add elements to a database is the [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) interface.

The [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) interface takes five arguments:

db

The database handle returned by [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html).

txnid

A transaction handle. In our simple case, we aren't expecting to recover the database after application or system crash, so we aren't using transactions, and will leave this argument NULL.

key

The key item for the key/data pair that we want to add to the database.

data

The data item for the key/data pair that we want to add to the database.

flags

Optional flags modifying the underlying behavior of the [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) interface.

Here's what the code to call [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) looks like:

#include <sys/types.h>

#include <stdio.h>

#include <db.h>

#define DATABASE "access.db"

int

main()

{

DB \*dbp;

**DBT key, data;**

int ret;

if ((ret = db\_create(&dbp, NULL, 0)) != 0) {

fprintf(stderr, "db\_create: %s\n", db\_strerror(ret));

exit (1);

}

if ((ret = dbp->open(dbp,

NULL, DATABASE, NULL, DB\_BTREE, DB\_CREATE, 0664)) != 0) {

dbp->err(dbp, ret, "%s", DATABASE);

goto err;

}

**memset(&key, 0, sizeof(key));**

**memset(&data, 0, sizeof(data));**

**key.data = "fruit";**

**key.size = sizeof("fruit");**

**data.data = "apple";**

**data.size = sizeof("apple");**

**if ((ret = dbp->put(dbp, NULL, &key, &data, 0)) == 0)**

**printf("db: %s: key stored.\n", (char \*)key.data);**

**else {**

**dbp->err(dbp, ret, "DB->put");**

**goto err;**

**}**

The first thing to notice about this new code is that we clear the [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) structures that we're about to pass as arguments to Berkeley DB functions. This is very important, and being careful to do so will result in fewer errors in your programs. All Berkeley DB structures instantiated in the application and handed to Berkeley DB should be cleared before use, without exception. This is necessary so that future versions of Berkeley DB may add additional fields to the structures. If applications clear the structures before use, it will be possible for Berkeley DB to change those structures without requiring that the applications be rewritten to be aware of the changes.

Notice also that we're storing the trailing nul byte found in the C strings **"fruit"** and **"apple"** in both the key and data items, that is, the trailing nul byte is part of the stored key, and therefore has to be specified in order to access the data item. There is no requirement to store the trailing nul byte, it simply makes it easier for us to display strings that we've stored in programming languages that use nul bytes to terminate strings.

In many applications, it is important not to overwrite existing data. For example, we might not want to store the key/data pair **fruit/apple** if it already existed, for example, if the key/data pair **fruit/cherry** had been previously stored into the database.

This is easily accomplished by adding the [DB\_NOOVERWRITE](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html#DB_NOOVERWRITE) flag to the [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) call:

**if ((ret =**

**dbp->put(dbp, NULL, &key, &data, DB\_NOOVERWRITE)) == 0)**

**printf("db: %s: key stored.\n", (char \*)key.data);**

**else {**

**dbp->err(dbp, ret, "DB->put");**

**goto err;**

**}**

This flag causes the underlying database functions to not overwrite any previously existing key/data pair. (Note that the value of the previously existing data doesn't matter in this case. The only question is if a key/data pair already exists where the key matches the key that we are trying to store.)

Specifying [DB\_NOOVERWRITE](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html#DB_NOOVERWRITE) opens up the possibility of a new Berkeley DB return value from the [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) function, [DB\_KEYEXIST](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html#DB_KEYEXIST), which means we were unable to add the key/data pair to the database because the key already existed in the database. While the above sample code simply displays a message in this case:

DB->put: DB\_KEYEXIST: Key/data pair already exists

The following code shows an explicit check for this possibility:

**switch (ret =**

**dbp->put(dbp, NULL, &key, &data, DB\_NOOVERWRITE)) {**

**case 0:**

**printf("db: %s: key stored.\n", (char \*)key.data);**

**break;**

**case DB\_KEYEXIST:**

**printf("db: %s: key previously stored.\n",**

**(char \*)key.data);**

**break;**

**default:**

**dbp->err(dbp, ret, "DB->put");**

**goto err;**

**}**

**Retrieving elements from a database**

The simplest way to retrieve elements from a database is the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) interface.

The [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) interface takes the same five arguments that the [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) interface takes:

db

The database handle returned by [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html).

txnid

A transaction ID. In our simple case, we aren't expecting to recover the database after application or system crash, so we aren't using transactions, and will leave this argument NULL.

key

The key item for the key/data pair that we want to retrieve from the database.

data

The data item for the key/data pair that we want to retrieve from the database.

flags

Optional flags modifying the underlying behavior of the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) interface.

Here's what the code to call [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) looks like:

#include <sys/types.h>

#include <stdio.h>

#include <db.h>

#define DATABASE "access.db"

int

main()

{

DB \*dbp;

DBT key, data;

int ret;

if ((ret = db\_create(&dbp, NULL, 0)) != 0) {

fprintf(stderr, "db\_create: %s\n", db\_strerror(ret));

exit (1);

}

if ((ret = dbp->open(dbp,

NULL, DATABASE, NULL, DB\_BTREE, DB\_CREATE, 0664)) != 0) {

dbp->err(dbp, ret, "%s", DATABASE);

goto err;

}

memset(&key, 0, sizeof(key));

memset(&data, 0, sizeof(data));

key.data = "fruit";

key.size = sizeof("fruit");

data.data = "apple";

data.size = sizeof("apple");

if ((ret = dbp->put(dbp, NULL, &key, &data, 0)) == 0)

printf("db: %s: key stored.\n", (char \*)key.data);

else {

dbp->err(dbp, ret, "DB->put");

goto err;

}

**if ((ret = dbp->get(dbp, NULL, &key, &data, 0)) == 0)**

**printf("db: %s: key retrieved: data was %s.\n",**

**(char \*)key.data, (char \*)data.data);**

**else {**

**dbp->err(dbp, ret, "DB->get");**

**goto err;**

**}**

It is not usually necessary to clear the [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) structures passed to the Berkeley DB functions between calls. This is not always true, when some of the less commonly used flags for [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) structures are used. The[DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) manual page specified the details of those cases.

It is possible, of course, to distinguish between system errors and the key/data pair simply not existing in the database. There are three standard returns from [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html):

1. The call might be successful and the key found, in which case the return value will be 0.
2. The call might be successful, but the key not found, in which case the return value will be [DB\_NOTFOUND](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/program/errorret.html#DB_NOTFOUND).
3. The call might not be successful, in which case the return value will be a system error.

# Removing elements from a database

The simplest way to remove elements from a database is the [DB->del](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_del.html) interface.

The [DB->del](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_del.html) interface takes four of the same five arguments that the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) and [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html) interfaces take. The difference is that there is no need to specify a data item, as the delete operation is only interested in the key that you want to remove.

db

The database handle returned by [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html).

txnid

A transaction ID. In our simple case, we aren't expecting to recover the database after application or system crash, so we aren't using transactions, and will leave this argument unspecified.

key

The key item for the key/data pair that we want to delete from the database.

flags

Optional flags modifying the underlying behavior of the [DB->del](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_del.html) interface. There are currently no available flags for this interface, so the flags argument should always be set to 0.

Here's what the code to call [DB->del](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_del.html) looks like:

#include <sys/types.h>

#include <stdio.h>

#include <db.h>

#define DATABASE "access.db"

int

main()

{

DB \*dbp;

DBT key, data;

int ret;

if ((ret = db\_create(&dbp, NULL, 0)) != 0) {

fprintf(stderr, "db\_create: %s\n", db\_strerror(ret));

exit (1);

}

if ((ret = dbp->open(dbp,

NULL, DATABASE, NULL, DB\_BTREE, DB\_CREATE, 0664)) != 0) {

dbp->err(dbp, ret, "%s", DATABASE);

goto err;

}

memset(&key, 0, sizeof(key));

memset(&data, 0, sizeof(data));

key.data = "fruit";

key.size = sizeof("fruit");

data.data = "apple";

data.size = sizeof("apple");

if ((ret = dbp->put(dbp, NULL, &key, &data, 0)) == 0)

printf("db: %s: key stored.\n", (char \*)key.data);

else {

dbp->err(dbp, ret, "DB->put");

goto err;

}

if ((ret = dbp->get(dbp, NULL, &key, &data, 0)) == 0)

printf("db: %s: key retrieved: data was %s.\n",

(char \*)key.data, (char \*)data.data);

else {

dbp->err(dbp, ret, "DB->get");

goto err;

}

**if ((ret = dbp->del(dbp, NULL, &key, 0)) == 0)**

**printf("db: %s: key was deleted.\n", (char \*)key.data);**

**else {**

**dbp->err(dbp, ret, "DB->del");**

**goto err;**

**}**

After the [DB->del](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_del.html) call returns, the entry to which the key **fruit** refers has been removed from the database.

# Closing a database

The only other operation that we need for our simple example is closing the database, and cleaning up the DB handle.

It is necessary that the database be closed. The most important reason for this is that Berkeley DB runs on top of an underlying buffer cache. If the modified database pages are never explicitly flushed to disk and the database is never closed, changes made to the database may never make it out to disk, because they are held in the Berkeley DB cache. As the default behavior of the close function is to flush the Berkeley DB cache, closing the database will update the on-disk information.

The [DB->close](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_close.html) interface takes two arguments:

db

The database handle returned by [db\_create](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_create.html).

flags

Optional flags modifying the underlying behavior of the [DB->close](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_close.html) interface.

Here's what the code to call [DB->close](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_close.html) looks like:

#include <sys/types.h>

#include <stdio.h>

#include <db.h>

#define DATABASE "access.db"

int

main()

{

DB \*dbp;

DBT key, data;

**int ret, t\_ret;**

if ((ret = db\_create(&dbp, NULL, 0)) != 0) {

fprintf(stderr, "db\_create: %s\n", db\_strerror(ret));

exit (1);

}

if ((ret = dbp->open(dbp,

NULL, DATABASE, NULL, DB\_BTREE, DB\_CREATE, 0664)) != 0) {

dbp->err(dbp, ret, "%s", DATABASE);

goto err;

}

memset(&key, 0, sizeof(key));

memset(&data, 0, sizeof(data));

key.data = "fruit";

key.size = sizeof("fruit");

data.data = "apple";

data.size = sizeof("apple");

if ((ret = dbp->put(dbp, NULL, &key, &data, 0)) == 0)

printf("db: %s: key stored.\n", (char \*)key.data);

else {

dbp->err(dbp, ret, "DB->put");

goto err;

}

if ((ret = dbp->get(dbp, NULL, &key, &data, 0)) == 0)

printf("db: %s: key retrieved: data was %s.\n",

(char \*)key.data, (char \*)data.data);

else {

dbp->err(dbp, ret, "DB->get");

goto err;

}

if ((ret = dbp->del(dbp, NULL, &key, 0)) == 0)

printf("db: %s: key was deleted.\n", (char \*)key.data);

else {

dbp->err(dbp, ret, "DB->del");

goto err;

}

if ((ret = dbp->get(dbp, NULL, &key, &data, 0)) == 0)

printf("db: %s: key retrieved: data was %s.\n",

(char \*)key.data, (char \*)data.data);

else

dbp->err(dbp, ret, "DB->get");

**err: if ((t\_ret = dbp->close(dbp, 0)) != 0 && ret == 0)**

**ret = t\_ret;**

exit(ret);

}

Note that we do not necessarily overwrite the **ret** variable, as it may contain error return information from a previous Berkeley DB call.

# ACCESS METHOD CONFIG

# What are the available access methods?

Berkeley DB currently offers four access methods: Btree, Hash, Queue and Recno.

### Btree

The Btree access method is an implementation of a sorted, balanced tree structure. Searches, insertions, and deletions in the tree all take O(log base\_b N) time, where base\_b is the average number of keys per page, and N is the total number of keys stored. Often, inserting ordered data into Btree implementations results in pages that are only half-full. Berkeley DB makes ordered (or inverse ordered) insertion the best case, resulting in nearly full-page space utilization.

### Hash

The Hash access method data structure is an implementation of Extended Linear Hashing, as described in "Linear Hashing: A New Tool for File and Table Addressing", Witold Litwin, *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, 1980.

### Queue

The Queue access method stores fixed-length records with logical record numbers as keys. It is designed for fast inserts at the tail and has a special cursor consume operation that deletes and returns a record from the head of the queue. The Queue access method uses record level locking.

### Recno

The Recno access method stores both fixed and variable-length records with logical record numbers as keys, optionally backed by a flat text (byte stream) file.

# Selecting an access method

The Berkeley DB access method implementation unavoidably interacts with each application's data set, locking requirements and data access patterns. For this reason, one access method may result in dramatically better performance for an application than another one. Applications whose data could be stored using more than one access method may want to benchmark their performance using the different candidates.

One of the strengths of Berkeley DB is that it provides multiple access methods with nearly identical interfaces to the different access methods. This means that it is simple to modify an application to use a different access method. Applications can easily benchmark the different Berkeley DB access methods against each other for their particular data set and access pattern.

Most applications choose between using the Btree or Hash access methods or between using the Queue and Recno access methods, because each of the two pairs offer similar functionality.

### Hash or Btree?

The Hash and Btree access methods should be used when logical record numbers are not the primary key used for data access. (If logical record numbers are a secondary key used for data access, the Btree access method is a possible choice, as it supports simultaneous access by a key and a record number.)

Keys in Btrees are stored in sorted order and the relationship between them is defined by that sort order. For this reason, the Btree access method should be used when there is any locality of reference among keys. Locality of reference means that accessing one particular key in the Btree implies that the application is more likely to access keys near to the key being accessed, where "near" is defined by the sort order. For example, if keys are timestamps, and it is likely that a request for an 8AM timestamp will be followed by a request for a 9AM timestamp, the Btree access method is generally the right choice. Or, for example, if the keys are names, and the application will want to review all entries with the same last name, the Btree access method is again a good choice.

There is little difference in performance between the Hash and Btree access methods on small data sets, where all, or most of, the data set fits into the cache. However, when a data set is large enough that significant numbers of data pages no longer fit into the cache, then the Btree locality of reference described previously becomes important for performance reasons. For example, there is no locality of reference for the Hash access method, and so key "AAAAA" is as likely to be stored on the same database page with key "ZZZZZ" as with key "AAAAB". In the Btree access method, because items are sorted, key "AAAAA" is far more likely to be near key "AAAAB" than key "ZZZZZ". So, if the application exhibits locality of reference in its data requests, then the Btree page read into the cache to satisfy a request for key "AAAAA" is much more likely to be useful to satisfy subsequent requests from the application than the Hash page read into the cache to satisfy the same request. This means that for applications with locality of reference, the cache is generally much more effective for the Btree access method than the Hash access method, and the Btree access method will make many fewer I/O calls.

However, when a data set becomes even larger, the Hash access method can outperform the Btree access method. The reason for this is that Btrees contain more metadata pages than Hash databases. The data set can grow so large that metadata pages begin to dominate the cache for the Btree access method. If this happens, the Btree can be forced to do an I/O for each data request because the probability that any particular data page is already in the cache becomes quite small. Because the Hash access method has fewer metadata pages, its cache stays "hotter" longer in the presence of large data sets. In addition, once the data set is so large that both the Btree and Hash access methods are almost certainly doing an I/O for each random data request, the fact that Hash does not have to walk several internal pages as part of a key search becomes a performance advantage for the Hash access method as well.

Application data access patterns strongly affect all of these behaviors, for example, accessing the data by walking a cursor through the database will greatly mitigate the large data set behavior describe above because each I/O into the cache will satisfy a fairly large number of subsequent data requests.

In the absence of information on application data and data access patterns, for small data sets either the Btree or Hash access methods will suffice. For data sets larger than the cache, we normally recommend using the Btree access method. If you have truly large data, then the Hash access method may be a better choice. The [db\_stat](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_stat.html) utility is a useful tool for monitoring how well your cache is performing.

### Queue or Recno?

The Queue or Recno access methods should be used when logical record numbers are the primary key used for data access. The advantage of the Queue access method is that it performs record level locking and for this reason supports significantly higher levels of concurrency than the Recno access method. The advantage of the Recno access method is that it supports a number of additional features beyond those supported by the Queue access method, such as variable-length records and support for backing flat-text files.

Logical record numbers can be mutable or fixed: mutable, where logical record numbers can change as records are deleted or inserted, and fixed, where record numbers never change regardless of the database operation. It is possible to store and retrieve records based on logical record numbers in the Btree access method. However, those record numbers are always mutable, and as records are deleted or inserted, the logical record number for other records in the database will change. The Queue access method always runs in fixed mode, and logical record numbers never change regardless of the database operation. The Recno access method can be configured to run in either mutable or fixed mode.

In addition, the Recno access method provides support for databases whose permanent storage is a flat text file and the database is used as a fast, temporary storage area while the data is being read or modified.

# Logical record numbers

The Berkeley DB Btree, Queue and Recno access methods can operate on logical record numbers. Record numbers are 1-based, not 0-based, that is, the first record in a database is record number 1.

In all cases for the Queue and Recno access methods, and when calling the Btree access method using the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) and [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html) methods with the [DB\_SET\_RECNO](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html#DB_SET_RECNO) flag specified, the **data** field of the key [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) must be a pointer to a memory location of type **db\_recno\_t**, as typedef'd in the standard Berkeley DB include file. The **size** field of the key [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) should be the size of that type (for example, "sizeof(db\_recno\_t)" in the C programming language). The **db\_recno\_t** type is a 32-bit unsigned type, which limits the number of logical records in a Queue or Recno database, and the maximum logical record which may be directly retrieved from a Btree database, to 4,294,967,295.

Record numbers in Queue databases wrap around. When the tail of the queue reaches the maximum record number, the next record appended will be given record number 1. If the head of the queue ever catches up to the tail of the queue, the Berkeley DB interface will return the system error EFBIG.

Record numbers in Recno databases can be configured to run in either mutable or fixed mode: mutable, where logical record numbers change as records are deleted or inserted, and fixed, where record numbers never change regardless of the database operation. Record numbers in Queue databases are always fixed, and never change regardless of the database operation. Record numbers in Btree databases are always mutable, and as records are deleted or inserted, the logical record number for other records in the database can change. See [Logically renumbering records](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/am_conf/renumber.html) for more information.

Configuring Btree databases to support record numbers can severely limit the throughput of applications with multiple concurrent threads writing the database, because locations used to store record counts often become hot spots that many different threads all need to update. In the case of a Btree supporting duplicate data items, the logical record number refers to a key and all of its data items, as duplicate data items are not individually numbered.

The following is an example function that reads records from standard input and stores them into a Recno database. The function then uses a cursor to step through the database and display the stored records.

int

recno\_build(dbp)

DB \*dbp;

{

DBC \*dbcp;

DBT key, data;

db\_recno\_t recno;

u\_int32\_t len;

int ret;

char buf[1024];

/\* Insert records into the database. \*/

memset(&key, 0, sizeof(DBT));

memset(&data, 0, sizeof(DBT));

for (recno = 1;; ++recno) {

printf("record #%lu> ", (u\_long)recno);

fflush(stdout);

if (fgets(buf, sizeof(buf), stdin) == NULL)

break;

if ((len = strlen(buf)) <= 1)

continue;

key.data = &recno;

key.size = sizeof(recno);

data.data = buf;

data.size = len - 1;

switch (ret = dbp->put(dbp, NULL, &key, &data, 0)) {

case 0:

break;

default:

dbp->err(dbp, ret, "DB->put");

break;

}

}

printf("\n");

/\* Acquire a cursor for the database. \*/

if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {

dbp->err(dbp, ret, "DB->cursor");

return (1);

}

/\* Re-initialize the key/data pair. \*/

memset(&key, 0, sizeof(key));

memset(&data, 0, sizeof(data));

/\* Walk through the database and print out the key/data pairs. \*/

while ((ret = dbcp->c\_get(dbcp, &key, &data, DB\_NEXT)) == 0)

printf("%lu : %.\*s\n",

\*(u\_long \*)key.data, (int)data.size, (char \*)data.data);

if (ret != DB\_NOTFOUND)

dbp->err(dbp, ret, "DBcursor->get");

/\* Close the cursor. \*/

if ((ret = dbcp->c\_close(dbcp)) != 0) {

dbp->err(dbp, ret, "DBcursor->close");

return (1);

}

return (0);

}

# Selecting a page size

The size of the pages used in the underlying database can be specified by calling the [DB->set\_pagesize](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_pagesize.html) method. The minimum page size is 512 bytes and the maximum page size is 64K bytes, and must be a power of two. If no page size is specified by the application, a page size is selected based on the underlying filesystem I/O block size. (A page size selected in this way has a lower limit of 512 bytes and an upper limit of 16K bytes.)

There are several issues to consider when selecting a pagesize: overflow record sizes, locking, I/O efficiency, and recoverability.

First, the page size implicitly sets the size of an overflow record. Overflow records are key or data items that are too large to fit on a normal database page because of their size, and are therefore stored in overflow pages. Overflow pages are pages that exist outside of the normal database structure. For this reason, there is often a significant performance penalty associated with retrieving or modifying overflow records. Selecting a page size that is too small, and which forces the creation of large numbers of overflow pages, can seriously impact the performance of an application.

Second, in the Btree, Hash and Recno access methods, the finest-grained lock that Berkeley DB acquires is for a page. (The Queue access method generally acquires record-level locks rather than page-level locks.) Selecting a page size that is too large, and which causes threads or processes to wait because other threads of control are accessing or modifying records on the same page, can impact the performance of your application.

Third, the page size specifies the granularity of I/O from the database to the operating system. Berkeley DB will give a page-sized unit of bytes to the operating system to be scheduled for reading/writing from/to the disk. For many operating systems, there is an internal **block size** which is used as the granularity of I/O from the operating system to the disk. Generally, it will be more efficient for Berkeley DB to write filesystem-sized blocks to the operating system and for the operating system to write those same blocks to the disk.

Selecting a database page size smaller than the filesystem block size may cause the operating system to coalesce or otherwise manipulate Berkeley DB pages and can impact the performance of your application. When the page size is smaller than the filesystem block size and a page written by Berkeley DB is not found in the operating system's cache, the operating system may be forced to read a block from the disk, copy the page into the block it read, and then write out the block to disk, rather than simply writing the page to disk. Additionally, as the operating system is reading more data into its buffer cache than is strictly necessary to satisfy each Berkeley DB request for a page, the operating system buffer cache may be wasting memory.

Alternatively, selecting a page size larger than the filesystem block size may cause the operating system to read more data than necessary. On some systems, reading filesystem blocks sequentially may cause the operating system to begin performing read-ahead. If requesting a single database page implies reading enough filesystem blocks to satisfy the operating system's criteria for read-ahead, the operating system may do more I/O than is required.

Fourth, when using the Berkeley DB Transactional Data Store product, the page size may affect the errors from which your database can recover See [Berkeley DB Recoverability](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/transapp/reclimit.html) for more information.

# Selecting a cache size

The size of the cache used for the underlying database can be specified by calling the [DB->set\_cachesize](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_cachesize.html) method. Choosing a cache size is, unfortunately, an art. Your cache must be at least large enough for your working set plus some overlap for unexpected situations.

When using the Btree access method, you must have a cache big enough for the minimum working set for a single access. This will include a root page, one or more internal pages (depending on the depth of your tree), and a leaf page. If your cache is any smaller than that, each new page will force out the least-recently-used page, and Berkeley DB will re-read the root page of the tree anew on each database request.

If your keys are of moderate size (a few tens of bytes) and your pages are on the order of 4K to 8K, most Btree applications will be only three levels. For example, using 20 byte keys with 20 bytes of data associated with each key, a 8KB page can hold roughly 400 keys (or 200 key/data pairs), so a fully populated three-level Btree will hold 32 million key/data pairs, and a tree with only a 50% page-fill factor will still hold 16 million key/data pairs. We rarely expect trees to exceed five levels, although Berkeley DB will support trees up to 255 levels.

The rule-of-thumb is that cache is good, and more cache is better. Generally, applications benefit from increasing the cache size up to a point, at which the performance will stop improving as the cache size increases. When this point is reached, one of two things have happened: either the cache is large enough that the application is almost never having to retrieve information from disk, or, your application is doing truly random accesses, and therefore increasing size of the cache doesn't significantly increase the odds of finding the next requested information in the cache. The latter is fairly rare -- almost all applications show some form of locality of reference.

That said, it is important not to increase your cache size beyond the capabilities of your system, as that will result in reduced performance. Under many operating systems, tying down enough virtual memory will cause your memory and potentially your program to be swapped. This is especially likely on systems without unified OS buffer caches and virtual memory spaces, as the buffer cache was allocated at boot time and so cannot be adjusted based on application requests for large amounts of virtual memory.

For example, even if accesses are truly random within a Btree, your access pattern will favor internal pages to leaf pages, so your cache should be large enough to hold all internal pages. In the steady state, this requires at most one I/O per operation to retrieve the appropriate leaf page.

You can use the [db\_stat](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_stat.html) utility to monitor the effectiveness of your cache. The following output is excerpted from the output of that utility's **-m** option:

prompt: db\_stat -m

131072 Cache size (128K).

4273 Requested pages found in the cache (97%).

134 Requested pages not found in the cache.

18 Pages created in the cache.

116 Pages read into the cache.

93 Pages written from the cache to the backing file.

5 Clean pages forced from the cache.

13 Dirty pages forced from the cache.

0 Dirty buffers written by trickle-sync thread.

130 Current clean buffer count.

4 Current dirty buffer count.

The statistics for this cache say that there have been 4,273 requests of the cache, and only 116 of those requests required an I/O from disk. This means that the cache is working well, yielding a 97% cache hit rate. The [db\_stat](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_stat.html) utility will present these statistics both for the cache as a whole and for each file within the cache separately.

# Selecting a byte order

Database files created by Berkeley DB can be created in either little- or big-endian formats. The byte order used for the underlying database is specified by calling the [DB->set\_lorder](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_lorder.html) method. If no order is selected, the native format of the machine on which the database is created will be used.

Berkeley DB databases are architecture independent, and any format database can be used on a machine with a different native format. In this case, as each page that is read into or written from the cache must be converted to or from the host format, and databases with non-native formats will incur a performance penalty for the run-time conversion.

**It is important to note that the Berkeley DB access methods do no data conversion for application specified data. Key/data pairs written on a little-endian format architecture will be returned to the application exactly as they were written when retrieved on a big-endian format architecture.**

# Duplicate data items

The Btree and Hash access methods support the creation of multiple data items for a single key item. By default, multiple data items are not permitted, and each database store operation will overwrite any previous data item for that key. To configure Berkeley DB for duplicate data items, call the [DB->set\_flags](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html) method with the [DB\_DUP](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_DUP) flag. Only one copy of the key will be stored for each set of duplicate data items. If the Btree access method comparison routine returns that two keys compare equally, it is undefined which of the two keys will be stored and returned from future database operations.

By default, Berkeley DB stores duplicates in the order in which they were added, that is, each new duplicate data item will be stored after any already existing data items. This default behavior can be overridden by using the [DBcursor->c\_put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html) method and one of the [DB\_AFTER](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html#DB_AFTER), [DB\_BEFORE](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html#DB_BEFORE) [DB\_KEYFIRST](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html#DB_KEYFIRST) or [DB\_KEYLAST](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html#DB_KEYLAST) flags. Alternatively, Berkeley DB may be configured to sort duplicate data items.

When stepping through the database sequentially, duplicate data items will be returned individually, as a key/data pair, where the key item only changes after the last duplicate data item has been returned. For this reason, duplicate data items cannot be accessed using the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) method, as it always returns the first of the duplicate data items. Duplicate data items should be retrieved using the Berkeley DB cursor interface, [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html).

There is an interface flag that permits applications to request the following data item only if it **is** a duplicate data item of the current entry, see [DB\_NEXT\_DUP](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_NEXT_DUP) for more information. There is an interface flag that permits applications to request the following data item only if it **is not** a duplicate data item of the current entry, see [DB\_NEXT\_NODUP](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_NEXT_NODUP) and [DB\_PREV\_NODUP](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_PREV_NODUP) for more information.

It is also possible to maintain duplicate records in sorted order. Sorting duplicates will significantly increase performance when searching them and performing equality joins, common operations when using secondary indices. To configure Berkeley DB to sort duplicate data items, the application must call the [DB->set\_flags](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html) method with the [DB\_DUPSORT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_DUPSORT) flag (in addition to the [DB\_DUP](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_DUP) flag). In addition, a custom sorting function may be specified using the [DB->set\_dup\_compare](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_dup_compare.html) method. If the [DB\_DUPSORT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_DUPSORT) flag is given, but no comparison routine is specified, then Berkeley DB defaults to the same lexicographical sorting used for Btree keys, with shorter items collating before longer items.

If the duplicate data items are unsorted, applications may store identical duplicate data items, or, for those that just like the way it sounds, *duplicate duplicates*.

**In this release it is an error to attempt to store identical duplicate data items when duplicates are being stored in a sorted order.** This restriction is expected to be lifted in a future release. There is an interface flag that permits applications to disallow storing duplicate data items when the database has been configured for sorted duplicates, see [DB\_NODUPDATA](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html#DB_NODUPDATA) for more information. Applications not wanting to permit duplicate duplicates in databases configured for sorted duplicates should begin using the [DB\_NODUPDATA](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html#DB_NODUPDATA) flag immediately.

For further information on how searching and insertion behaves in the presence of duplicates (sorted or not), see the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html), [DB->put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_put.html), [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html) and [DBcursor->c\_put](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_put.html) documentation.

# Non-local memory allocation

Berkeley DB allocates memory for returning key/data pairs and statistical information which becomes the responsibility of the application. There are also interfaces where an application will allocate memory which becomes the responsibility of Berkeley DB.

On systems in which there may be multiple library versions of the standard allocation routines (notably Windows NT), transferring memory between the library and the application will fail because the Berkeley DB library allocates memory from a different heap than the application uses to free it, or vice versa. To avoid this problem, the [DB\_ENV->set\_alloc](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/env_set_alloc.html) and [DB->set\_alloc](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_alloc.html) methods can be used to give Berkeley

# Btree comparison

The Btree data structure is a sorted, balanced tree structure storing associated key/data pairs. By default, the sort order is lexicographical, with shorter keys collating before longer keys. The user can specify the sort order for the Btree by using the [DB->set\_bt\_compare](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_bt_compare.html) method.

Sort routines are passed pointers to keys as arguments. The keys are represented as [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) structures. The routine must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second argument. The only fields that the routines may examine in the [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) structures are **data** and **size** fields.

An example routine that might be used to sort integer keys in the database is as follows:

int

compare\_int(dbp, a, b)

DB \*dbp;

const DBT \*a, \*b;

{

int ai, bi;

/\*

\* Returns:

\* < 0 if a < b

\* = 0 if a = b

\* > 0 if a > b

\*/

memcpy(&ai, a->data, sizeof(int));

memcpy(&bi, b->data, sizeof(int));

return (ai - bi);

}

Note that the data must first be copied into memory that is appropriately aligned, as Berkeley DB does not guarantee any kind of alignment of the underlying data, including for comparison routines. When writing comparison routines, remember that databases created on machines of different architectures may have different integer byte orders, for which your code may need to compensate.

An example routine that might be used to sort keys based on the first five bytes of the key (ignoring any subsequent bytes) is as follows:

int

compare\_dbt(dbp, a, b)

DB \*dbp;

const DBT \*a, \*b;

{

int len;

u\_char \*p1, \*p2;

/\*

\* Returns:

\* < 0 if a < b

\* = 0 if a = b

\* > 0 if a > b

\*/

for (p1 = a->data, p2 = b->data, len = 5; len--; ++p1, ++p2)

if (\*p1 != \*p2)

return ((long)\*p1 - (long)\*p2);

return (0);

}

All comparison functions must cause the keys in the database to be well-ordered. The most important implication of being well-ordered is that the key relations must be transitive, that is, if key A is less than key B, and key B is less than key C, then the comparison routine must also return that key A is less than key C. In addition, comparisons will only be able to return 0 when comparing full length keys; partial key comparisons must always return a result less than or greater than 0.

It is reasonable for a comparison function to not examine an entire key in some applications, which implies that partial keys may be specified to the Berkeley DB interfaces. When partial keys are specified to Berkeley DB, interfaces which retrieve data items based on a user-specified key (for example, [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) and [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html) with the [DB\_SET](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_SET) flag), will not modify the user-specified key by returning the actual key stored in the database. The actual key can be retrieved by calling the [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html) method with the [DB\_CURRENT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_CURRENT) flag.

# Btree prefix comparison

The Berkeley DB Btree implementation maximizes the number of keys that can be stored on an internal page by storing only as many bytes of each key as are necessary to distinguish it from adjacent keys. The prefix comparison routine is what determines this minimum number of bytes (that is, the length of the unique prefix), that must be stored. A prefix comparison function for the Btree can be specified by calling [DB->set\_bt\_prefix](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_bt_prefix.html).

The prefix comparison routine must be compatible with the overall comparison function of the Btree, since what distinguishes any two keys depends entirely on the function used to compare them. This means that if a prefix comparison routine is specified by the application, a compatible overall comparison routine must also have been specified.

Prefix comparison routines are passed pointers to keys as arguments. The keys are represented as [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html) structures. The prefix comparison function must return the number of bytes of the second key argument that are necessary to determine if it is greater than the first key argument. If the keys are equal, the length of the second key should be returned. The only fields that the routines may examine in the [DBT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbt_class.html)structures are **data** and **size** fields.

An example prefix comparison routine follows:

u\_int32\_t

compare\_prefix(dbp, a, b)

DB \*dbp;

const DBT \*a, \*b;

{

size\_t cnt, len;

u\_int8\_t \*p1, \*p2;

cnt = 1;

len = a->size > b->size ? b->size : a->size;

for (p1 =

a->data, p2 = b->data; len--; ++p1, ++p2, ++cnt)

if (\*p1 != \*p2)

return (cnt);

/\*

\* They match up to the smaller of the two sizes.

\* Collate the longer after the shorter.

\*/

if (a->size < b->size)

return (a->size + 1);

if (b->size < a->size)

return (b->size + 1);

return (b->size);

}

The usefulness of this functionality is data-dependent, but in some data sets can produce significantly reduced tree sizes and faster search times.

# Minimum keys per page

The number of keys stored on each page affects the size of a Btree and how it is maintained. Therefore, it also affects the retrieval and search performance of the tree. For each Btree, Berkeley DB computes a maximum key and data size. This size is a function of the page size and the fact that at least two key/data pairs must fit on any Btree page. Whenever key or data items exceed the calculated size, they are stored on overflow pages instead of in the standard Btree leaf pages.

Applications may use the [DB->set\_bt\_minkey](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_bt_minkey.html) method to change the minimum number of keys that must fit on a Btree page from two to another value. Altering this value in turn alters the on-page maximum size, and can be used to force key and data items which would normally be stored in the Btree leaf pages onto overflow pages.

Some data sets can benefit from this tuning. For example, consider an application using large page sizes, with a data set almost entirely consisting of small key and data items, but with a few large items. By setting the minimum number of keys that must fit on a page, the application can force the outsized items to be stored on overflow pages. That in turn can potentially keep the tree more compact, that is, with fewer internal levels to traverse during searches.

The following calculation is similar to the one performed by the Btree implementation. (The **minimum\_keys** value is multiplied by 2 because each key/data pair requires 2 slots on a Btree page.)

maximum\_size = page\_size / (minimum\_keys \* 2)

Using this calculation, if the page size is 8KB and the default **minimum\_keys** value of 2 is used, then any key or data items larger than 2KB will be forced to an overflow page. If an application were to specify a**minimum\_key** value of 100, then any key or data items larger than roughly 40 bytes would be forced to overflow pages.

It is important to remember that accesses to overflow pages do not perform as well as accesses to the standard Btree leaf pages, and so setting the value incorrectly can result in overusing overflow pages and decreasing the application's overall performance.

# Retrieving Btree records by logical record number

The Btree access method optionally supports retrieval by logical record numbers. To configure a Btree to support record numbers, call the [DB->set\_flags](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html) method with the [DB\_RECNUM](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_RECNUM) flag.

Configuring a Btree for record numbers should not be done lightly. While often useful, it may significantly slow down the speed at which items can be stored into the database, and can severely impact application throughput. Generally it should be avoided in trees with a need for high write concurrency.

To retrieve by record number, use the [DB\_SET\_RECNO](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html#DB_SET_RECNO) flag to the [DB->get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_get.html) and [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html) methods. The following is an example of a routine that displays the data item for a Btree database created with the [DB\_RECNUM](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_RECNUM) option.

int

rec\_display(dbp, recno)

DB \*dbp;

db\_recno\_t recno;

{

DBT key, data;

int ret;

memset(&key, 0, sizeof(key));

key.data = &recno;

key.size = sizeof(recno);

memset(&data, 0, sizeof(data));

if ((ret = dbp->get(dbp, NULL, &key, &data, DB\_SET\_RECNO)) != 0)

return (ret);

printf("data for %lu: %.\*s\n",

(u\_long)recno, (int)data.size, (char \*)data.data);

return (0);

}

To determine a key's record number, use the [DB\_GET\_RECNO](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_GET_RECNO) flag to the [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html) method. The following is an example of a routine that displays the record number associated with a specific key.

int

recno\_display(dbp, keyvalue)

DB \*dbp;

char \*keyvalue;

{

DBC \*dbcp;

DBT key, data;

db\_recno\_t recno;

int ret, t\_ret;

/\* Acquire a cursor for the database. \*/

if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {

dbp->err(dbp, ret, "DB->cursor");

goto err;

}

/\* Position the cursor. \*/

memset(&key, 0, sizeof(key));

key.data = keyvalue;

key.size = strlen(keyvalue);

memset(&data, 0, sizeof(data));

if ((ret = dbcp->c\_get(dbcp, &key, &data, DB\_SET)) != 0) {

dbp->err(dbp, ret, "DBC->c\_get(DB\_SET): %s", keyvalue);

goto err;

}

/\*

\* Request the record number, and store it into appropriately

\* sized and aligned local memory.

\*/

memset(&data, 0, sizeof(data));

data.data = &recno;

data.ulen = sizeof(recno);

data.flags = DB\_DBT\_USERMEM;

if ((ret = dbcp->c\_get(dbcp, &key, &data, DB\_GET\_RECNO)) != 0) {

dbp->err(dbp, ret, "DBC->c\_get(DB\_GET\_RECNO)");

goto err;

}

printf("key for requested key was %lu\n", (u\_long)recno);

err: /\* Close the cursor. \*/

if ((t\_ret = dbcp->c\_close(dbcp)) != 0) {

if (ret == 0)

ret = t\_ret;

dbp->err(dbp, ret, "DBC->close");

}

return (ret);

}

# Page fill factor

The density, or page fill factor, is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. If you know the average sizes of the keys and data in your data set, setting the fill factor can enhance performance. A reasonable rule to use to compute fill factor is:

(pagesize - 32) / (average\_key\_size + average\_data\_size + 8)

The desired density within the hash table can be specified by calling the [DB->set\_h\_ffactor](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_h_ffactor.html) method. If no density is specified, one will be selected dynamically as pages are filled.

# Specifying a database hash

The database hash determines in which bucket a particular key will reside. The goal of hashing keys is to distribute keys equally across the database pages, therefore it is important that the hash function work well with the specified keys so that the resulting bucket usage is relatively uniform. A hash function that does not work well can effectively turn into a sequential list.

No hash performs equally well on all possible data sets. It is possible that applications may find that the default hash function performs poorly with a particular set of keys. The distribution resulting from the hash function can be checked using [db\_stat](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_stat.html) utility. By comparing the number of hash buckets and the number of keys, one can decide if the entries are hashing in a well-distributed manner.

The hash function for the hash table can be specified by calling the [DB->set\_h\_hash](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_h_hash.html) method. If no hash function is specified, a default function will be used. Any application-specified hash function must take a reference to a [DB](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_class.html) object, a pointer to a byte string and its length, as arguments and return an unsigned, 32-bit hash value.

# Hash table size

When setting up the hash database, knowing the expected number of elements that will be stored in the hash table is useful. This value can be used by the Hash access method implementation to more accurately construct the necessary number of buckets that the database will eventually require.

The anticipated number of elements in the hash table can be specified by calling the [DB->set\_h\_nelem](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_h_nelem.html) method. If not specified, or set too low, hash tables will expand gracefully as keys are entered, although a slight performance degradation may be noticed. In order for the estimated number of elements to be a useful value to Berkeley DB, the [DB->set\_h\_ffactor](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_h_ffactor.html) method must also be called to set the page fill factor.

# Managing record-based databases

When using fixed- or variable-length record-based databases, particularly with flat-text backing files, there are several items that the user can control. The Recno access method can be used to store either variable- or fixed-length data items. By default, the Recno access method stores variable-length data items. The Queue access method can only store fixed-length data items.

### Record Delimiters

When using the Recno access method to store variable-length records, records read from any backing source file are separated by a specific byte value which marks the end of one record and the beginning of the next. This delimiting value is ignored except when reading records from a backing source file, that is, records may be stored into the database that include the delimiter byte. However, if such records are written out to the backing source file and the backing source file is subsequently read into a database, the records will be split where delimiting bytes were found.

For example, UNIX text files can usually be interpreted as a sequence of variable-length records separated by ASCII newline characters. This byte value (ASCII 0x0a) is the default delimiter. Applications may specify a different delimiting byte using the [DB->set\_re\_delim](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_re_delim.html) interface. If no backing source file is being used, there is no reason to set the delimiting byte value.

### Record Length

When using the Recno or Queue access methods to store fixed-length records, the record length must be specified. Since the Queue access method always uses fixed-length records, the user must always set the record length prior to creating the database. Setting the record length is what causes the Recno access method to store fixed-length, not variable-length, records.

The length of the records is specified by calling the [DB->set\_re\_len](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_re_len.html) method. The default length of the records is 0 bytes. Any record read from a backing source file or otherwise stored in the database that is shorter than the declared length will automatically be padded as described for the [DB->set\_re\_pad](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_re_pad.html) method. Any record stored that is longer than the declared length results in an error. For further information on backing source files, see [Flat-text backing files](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/am_conf/re_source.html).

### Record Padding Byte Value

When storing fixed-length records in a Queue or Recno database, a pad character may be specified by calling the [DB->set\_re\_pad](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_re_pad.html) method. Any record read from the backing source file or otherwise stored in the database that is shorter than the expected length will automatically be padded with this byte value. If fixed-length records are specified but no pad value is specified, a space character (0x20 in the ASCII character set) will be used. For further information on backing source files, see [Flat-text backing files](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/am_conf/re_source.html).

# Selecting a Queue extent size

In Queue databases, records are allocated sequentially and directly mapped to an offset within the file storage for the database. As records are deleted from the Queue, pages will become empty and will not be reused in normal queue operations. To facilitate the reclamation of disk space a Queue may be partitioned into extents. Each extent is kept in a separate physical file.

Extent files are automatically created as needed and marked for deletion when the head of the queue moves off the extent. The extent will not be deleted until all processes close the extent. In addition, Berkeley DB caches a small number of extents that have been recently used; this may delay when an extent will be deleted. The number of extents left open depends on queue activity.

The extent size specifies the number of pages that make up each extent. By default, if no extent size is specified, the Queue resides in a single file and disk space is not reclaimed. In choosing an extent size there is a tradeoff between the amount of disk space used and the overhead of creating and deleting files. If the extent size is too small, the system will pay a performance penalty, creating and deleting files frequently. In addition, if the active part of the queue spans many files, all those files will need to be open at the same time, consuming system and process file resources.

# Flat-text backing files

It is possible to back any Recno database (either fixed or variable length) with a flat-text source file. This provides fast read (and potentially write) access to databases that are normally created and stored as flat-text files. The backing source file may be specified by calling the [DB->set\_re\_source](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_re_source.html) method.

The backing source file will be read to initialize the database. In the case of variable length records, the records are assumed to be separated as described for the [DB->set\_re\_delim](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_re_delim.html) method interface. For example, standard UNIX byte stream files can be interpreted as a sequence of variable length records separated by ASCII newline characters. This is the default.

When cached data would normally be written back to the underlying database file (for example, when the [DB->close](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_close.html) or [DB->sync](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_sync.html) methods are called), the in-memory copy of the database will be written back to the backing source file.

The backing source file must already exist (but may be zero-length) when [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) is called. By default, the backing source file is read lazily, that is, records are not read from the backing source file until they are requested by the application. If multiple processes (not threads) are accessing a Recno database concurrently and either inserting or deleting records, the backing source file must be read in its entirety before more than a single process accesses the database, and only that process should specify the backing source file as part of the [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) call. This can be accomplished by calling the [DB->set\_flags](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html) method with the [DB\_SNAPSHOT](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_SNAPSHOT) flag.

Reading and writing the backing source file cannot be transactionally protected because it involves filesystem operations that are not part of the Berkeley DB transaction methodology. For this reason, if a temporary database is used to hold the records (a NULL was specified as the file argument to [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html)), **it is possible to lose the contents of the backing source file if the system crashes at the right instant**. If a permanent file is used to hold the database (a filename was specified as the file argument to [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html)), normal database recovery on that file can be used to prevent information loss. It is still possible that the contents of the backing source file itself will be corrupted or lost if the system crashes.

For all of the above reasons, the backing source file is generally used to specify databases that are read-only for Berkeley DB applications, and that are either generated on the fly by software tools, or modified using a different mechanism such as a text editor.

# Logically renumbering records

Records stored in the Queue and Recno access methods are accessed by logical record number. In all cases in Btree databases, and optionally in Recno databases (see the [DB->set\_flags](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html) method and the[DB\_RENUMBER](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_set_flags.html#DB_RENUMBER) flag for more information), record numbers are mutable. This means that the record numbers may change as records are added to and deleted from the database. The deletion of record number 4 causes any records numbered 5 and higher to be renumbered downward by 1; the addition of a new record after record number 4 causes any records numbered 5 and higher to be renumbered upward by 1. In all cases in Queue databases, and by default in Recno databases, record numbers are not mutable, and the addition or deletion of records to the database will not cause already-existing record numbers to change. For this reason, new records cannot be inserted between already-existing records in databases with immutable record numbers.

Cursors pointing into a Btree database or a Recno database with mutable record numbers maintain a reference to a specific record, rather than a record number, that is, the record they reference does not change as other records are added or deleted. For example, if a database contains three records with the record numbers 1, 2, and 3, and the data items "A", "B", and "C", respectively, the deletion of record number 2 ("B") will cause the record "C" to be renumbered downward to record number 2. A cursor positioned at record number 3 ("C") will be adjusted and continue to point to "C" after the deletion. Similarly, a cursor previously referring to the now deleted record number 2 will be positioned between the new record numbers 1 and 2, and an insertion using that cursor will appear between those records. In this manner records can be added and deleted to a database without disrupting the sequential traversal of the database by a cursor.

Only cursors created using a single [DB](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_class.html) handle can adjust each other's position in this way, however. If multiple [DB](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_class.html) handles have a renumbering Recno database open simultaneously (as when multiple processes share a single database environment), a record referred to by one cursor could change underfoot if a cursor created using another [DB](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_class.html) handle inserts or deletes records into the database. For this reason, applications using Recno databases with mutable record numbers will usually make all accesses to the database using a single [DB](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_class.html) handle and cursors created from that handle, or will otherwise single-thread access to the database, for example, by using the Berkeley DB Concurrent Data Store product.

In any Queue or Recno databases, creating new records will cause the creation of multiple records if the record number being created is more than one greater than the largest record currently in the database. For example, creating record number 28, when record 25 was previously the last record in the database, will implicitly create records 26 and 27 as well as 28. All first, last, next and previous cursor operations will automatically skip over these implicitly created records. So, if record number 5 is the only record the application has created, implicitly creating records 1 through 4, the [DBcursor->c\_get](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html)interface with the [DB\_FIRST](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbc_get.html#DB_FIRST) flag will return record number 5, not record number 1. Attempts to explicitly retrieve implicitly created records by their record number will result in a special error return,[DB\_KEYEMPTY](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/program/errorret.html#DB_KEYEMPTY).

In any Berkeley DB database, attempting to retrieve a deleted record, using a cursor positioned on the record, results in a special error return, [DB\_KEYEMPTY](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/program/errorret.html#DB_KEYEMPTY). In addition, when using Queue databases or Recno databases with immutable record numbers, attempting to retrieve a deleted record by its record number will also result in the [DB\_KEYEMPTY](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/program/errorret.html#DB_KEYEMPTY) return.

# 6. Berkeley DB architecture

# The big picture

The previous chapters in this Reference Guide have described applications that use the Berkeley DB access methods for fast data storage and retrieval. The applications described in the following chapters are similar in nature to the access method applications, but they are also threaded and/or recoverable in the face of application or system failure.

Application code that uses only the Berkeley DB access methods might appear as follows:

switch (ret = dbp->put(dbp, NULL, &key, &data, 0)) {

case 0:

printf("db: %s: key stored.\n", (char \*)key.data);

break;

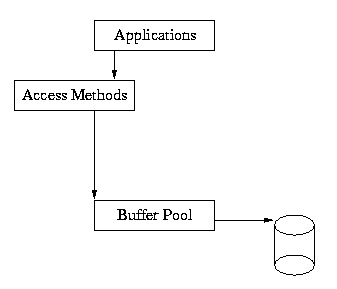
default:

dbp->err(dbp, ret, "dbp->put");

exit (1);

}

The underlying Berkeley DB architecture that supports this is



As you can see from this diagram, the application makes calls into the access methods, and the access methods use the underlying shared memory buffer cache to hold recently used file pages in main memory.

When applications require recoverability, their calls to the Access Methods must be wrapped in calls to the transaction subsystem. The application must inform Berkeley DB where to begin and end transactions, and must be prepared for the possibility that an operation may fail at any particular time, causing the transaction to abort.

An example of transaction-protected code might appear as follows:

for (fail = 0;;) {

/\* Begin the transaction. \*/

if ((ret = dbenv->txn\_begin(dbenv, NULL, &tid, 0)) != 0) {

dbenv->err(dbenv, ret, "dbenv->txn\_begin");

exit (1);

}

/\* Store the key. \*/

switch (ret = dbp->put(dbp, tid, &key, &data, 0)) {

case 0:

/\* Success: commit the change. \*/

printf("db: %s: key stored.\n", (char \*)key.data);

if ((ret = tid->commit(tid, 0)) != 0) {

dbenv->err(dbenv, ret, "DB\_TXN->commit");

exit (1);

}

return (0);

case DB\_LOCK\_DEADLOCK:

default:

/\* Failure: retry the operation. \*/

if ((t\_ret = tid->abort(tid)) != 0) {

dbenv->err(dbenv, t\_ret, "DB\_TXN->abort");

exit (1);

}

if (++fail == MAXIMUM\_RETRY)

return (ret);

continue;

}

}

In this example, the same operation is being done as before; however, it is wrapped in transaction calls. The transaction is started with [DB\_ENV->txn\_begin](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/txn_begin.html) and finished with [DB\_TXN->commit](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/txn_commit.html). If the operation fails due to a deadlock, the transaction is aborted using [DB\_TXN->abort](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/txn_abort.html), after which the operation may be retried.

There are actually five major subsystems in Berkeley DB, as follows:

Access Methods

The access methods subsystem provides general-purpose support for creating and accessing database files formatted as Btrees, Hashed files, and Fixed- and Variable-length records. These modules are useful in the absence of transactions for applications that need fast formatted file support. See [DB->open](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_open.html) and [DB->cursor](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/db_cursor.html) for more information. These functions were already discussed in detail in the previous chapters.

Memory Pool

The Memory Pool subsystem is the general-purpose shared memory buffer pool used by Berkeley DB. This is the shared memory cache that allows multiple processes and threads within processes to share access to databases. This module is useful outside of the Berkeley DB package for processes that require portable, page-oriented, cached, shared file access.

Transaction

The Transaction subsystem allows a group of database changes to be treated as an atomic unit so that either all of the changes are done, or none of the changes are done. The transaction subsystem implements the Berkeley DB transaction model. This module is useful outside of the Berkeley DB package for processes that want to transaction-protect their own data modifications.

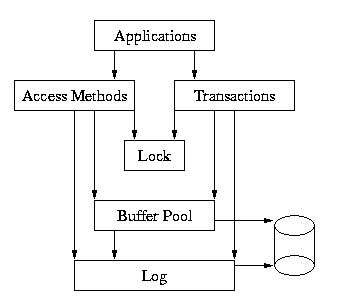
Locking

The Locking subsystem is the general-purpose lock manager used by Berkeley DB. This module is useful outside of the Berkeley DB package for processes that require a portable, fast, configurable lock manager.

Logging

The Logging subsystem is the write-ahead logging used to support the Berkeley DB transaction model. It is largely specific to the Berkeley DB package, and unlikely to be useful elsewhere except as a supporting module for the Berkeley DB transaction subsystem.

Here is a more complete picture of the Berkeley DB library:



In this model, the application makes calls to the access methods and to the Transaction subsystem. The access methods and Transaction subsystems in turn make calls into the Memory Pool, Locking and Logging subsystems on behalf of the application.

The underlying subsystems can be used independently by applications. For example, the Memory Pool subsystem can be used apart from the rest of Berkeley DB by applications simply wanting a shared memory buffer pool, or the Locking subsystem may be called directly by applications that are doing their own locking outside of Berkeley DB. However, this usage is not common, and most applications will either use only the access methods subsystem, or the access methods subsystem wrapped in calls to the Berkeley DB transaction interfaces.

# Programming model

Berkeley DB is a database library, in which the library is linked into the address space of the application using it. The code using Berkeley DB may be a standalone application or it may be a server providing functionality to many clients via inter-process or remote-process communication (IPC/RPC).

In the standalone application model, one or more applications link the Berkeley DB library directly into their address spaces. There may be many threads of control in this model because Berkeley DB supports locking for both multiple processes and for multiple threads within a process. This model provides significantly faster access to the database functionality, but implies trust among all threads of control sharing the database environment because they will have the ability to read, write and potentially corrupt each other's data.

In the client-server model, developers write a database server application that accepts requests via some form of IPC/RPC, and issues calls to the Berkeley DB interfaces based on those requests. In this model, the database server is the only application linking the Berkeley DB library into its address space. The client-server model trades performance for protection because it does not require that the applications share a protection domain with the server, but IPC/RPC is slower than a function call. Of course, this model also greatly simplifies the creation of network client-server applications.

|  |
| --- |
|  |

# Programmatic APIs

The Berkeley DB subsystems can be accessed through interfaces from multiple languages. The standard library interface is ANSI C. Applications can also use Berkeley DB via C++ or Java, as well as from scripting languages. Environments can be shared among applications written by using any of theses APIs. For example, you might have a local server written in C or C++, a script for an administrator written in Perl or Tcl, and a Web-based user interface written in Java -- all sharing a single database environment.

### C

The Berkeley DB library is written entirely in ANSI C. C applications use a single include file:

#include <db.h>

### C++

The C++ classes provide a thin wrapper around the C API, with the major advantages being improved encapsulation and an optional exception mechanism for errors. C++ applications use a single include file:

#include <db\_cxx.h>

The classes and methods are named in a fashion that directly corresponds to structures and functions in the C interface. Likewise, arguments to methods appear in the same order as the C interface, except to remove the explicit **this** pointer. The #defines used for flags are identical between the C and C++ interfaces.

As a rule, each C++ object has exactly one structure from the underlying C API associated with it. The C structure is allocated with each constructor call and deallocated with each destructor call. Thus, the rules the user needs to follow in allocating and deallocating structures are the same between the C and C++ interfaces.

To ensure portability to many platforms, both new and old, Berkeley DB makes as few assumptions as possible about the C++ compiler and library. For example, it does not expect STL, templates, or namespaces to be available. The newest C++ feature used is exceptions, which are used liberally to transmit error information. Even the use of exceptions can be disabled at runtime.

### Java

The Java classes provide a layer around the C API that is almost identical to the C++ layer. The classes and methods are, for the most part identical to the C++ layer. Berkeley DB constants and #defines are represented as "static final int" values. Error conditions are communicated as Java exceptions.

As in C++, each Java object has exactly one structure from the underlying C API associated with it. The Java structure is allocated with each constructor or open call, but is deallocated only by the Java garbage collector. Because the timing of garbage collection is not predictable, applications should take care to do a close when finished with any object that has a close method.

### Dbm/Ndbm, Hsearch

Berkeley DB supports the standard UNIX interfaces [dbm](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbm.html), [ndbm](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbm.html), and [hsearch](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/hsearch.html). After including a new header file and recompiling, programs will run orders of magnitude faster, and underlying databases can grow as large as necessary. Also, historic [dbm](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbm.html) and [ndbm](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/api_c/dbm.html) applications can fail once some number of entries are inserted into the database, in which the number depends on the effectiveness of the internal hashing function on the particular data set. This is not a problem with Berkeley DB.

# Scripting languages

### Perl

Two Perl APIs are distributed with the Berkeley DB release. The Perl interface to Berkeley DB version 1.85 is called DB\_File. The Perl interface to Berkeley DB version 2 and later is called BerkeleyDB. See[Using Berkeley DB with Perl](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/perl/intro.html) for more information.

### Tcl

A Tcl API is distributed with the Berkeley DB release. See [Using Berkeley DB with Tcl](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/ref/tcl/intro.html) for more information.

# Supporting utilities

The following are the standalone utilities that provide supporting functionality for the Berkeley DB environment:

[berkeley\_db\_svc](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/berkeley_db_svc.html)

The [berkeley\_db\_svc](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/berkeley_db_svc.html) utility is the Berkeley DB RPC server that provides standard server functionality for client applications.

[db\_archive](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_archive.html)

The [db\_archive](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_archive.html) utility supports database backup and archival, and log file administration. It facilitates log reclamation and the creation of database snapshots. Generally, some form of log archival must be done if a database environment has been configured for logging or transactions.

[db\_checkpoint](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_checkpoint.html)

The [db\_checkpoint](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_checkpoint.html) utility runs as a daemon process, monitoring the database log and periodically issuing checkpoints. It facilitates log reclamation and the creation of database snapshots. Generally, some form of database checkpointing must be done if a database environment has been configured for transactions.

[db\_deadlock](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_deadlock.html)

The [db\_deadlock](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_deadlock.html) utility runs as a daemon process, periodically traversing the database lock structures and aborting transactions when it detects a deadlock. Generally, some form of deadlock detection must be done if a database environment has been configured for locking.

[db\_dump](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_dump.html)

The [db\_dump](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_dump.html) utility writes a copy of the database to a flat-text file in a portable format.

[db\_load](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_load.html)

The [db\_load](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_load.html) utility reads the flat-text file produced by [db\_dump](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_dump.html) and loads it into a database file.

[db\_printlog](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_printlog.html)

The [db\_printlog](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_printlog.html) utility displays the contents of Berkeley DB log files in a human-readable and parsable format.

[db\_recover](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_recover.html)

The [db\_recover](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_recover.html) utility runs after an unexpected Berkeley DB or system failure to restore the database to a consistent state. Generally, some form of database recovery must be done if databases are being modified.

[db\_stat](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_stat.html)

The [db\_stat](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_stat.html) utility displays statistics for databases and database environments.

[db\_upgrade](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_upgrade.html)

The [db\_upgrade](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_upgrade.html) utility provides a command-line interface for upgrading underlying database formats.

[db\_verify](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_verify.html)

The [db\_verify](https://www.stanford.edu/class/cs276a/projects/docs/berkeleydb/utility/db_verify.html) utility provides a command-line interface for verifying the database format.

All of the functionality implemented for these utilities is also available as part of the standard Berkeley DB API. This means that threaded applications can easily create a thread that calls the same Berkeley DB functions as do the utilities. This often simplifies an application environment by removing the necessity for multiple processes to negotiate database and database environment creation and shut down.