Oracle Berkeley DB

Programmer's Reference Guide

Release 4.8

Legal Notice

Published 4/12/2010

Table of Contents

Preface

Welcome to Berkeley DB (DB). This document provides an introduction and usage notes for skilled programmers who wish to use the Berkeley DB APIs.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Structure names are represented in `monospaced font`, as are `method names`. For example: "`DB->open()` is a method on a `DB` handle."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DB_INSTALL* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

```
/* File: gettingstarted_common.h */
typedef struct stock_dbs {
    DB *inventory_dbp; /* Database containing inventory information */
    DB *vendor_dbp;    /* Database containing vendor information */

    char *db_home_dir;       /* Directory containing the database files */
    char *inventory_db_name; /* Name of the inventory database */
    char *vendor_db_name;    /* Name of the vendor database */
} STOCK_DBS;
```

Finally, notes of interest are represented using a note block such as this.

For More Information

Beyond this manual, you may also find the following sources of information useful when building a DB application:

- Getting Started with Transaction Processing for C [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_txn/C/BerkeleyDB-Core-C-Txn.pdf]
- Berkeley DB Getting Started with Replicated Applications for C [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_db_rep/C/Replication_C_GSG.pdf]
- Berkeley DB C API [http://www.oracle.com/technology/documentation/berkeley-db/db/api_reference/C/BDB-C_APIReference.pdf]
- Berkeley DB C++ API [http://www.oracle.com/technology/documentation/berkeley-db/db/api_reference/CXX/BDB-CXX_APIReference.pdf]
- Berkeley DB STL API [http://www.oracle.com/technology/documentation/berkeley-db/db/api_reference/STL/BDB-STL_APIReference.pdf]

- Berkeley DB TCL API [http://www.oracle.com/technology/documentation/berkeley-db/db/api_reference/TCL/BDB-TCL_APIReference.pdf]
- Berkeley DB Programmer's Reference Guide [http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/BDB_Prog_Reference.pdf]

Chapter 1. Introduction

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 shift 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 Berkeley DB is distributed under a license that conforms to the Open Source Definition [http://www.opensource.org/osd.html]. This license guarantees Berkeley DB is freely available for use and redistribution in other Open Source applications. Oracle Corporation sells commercial licenses allowing the redistribution of Berkeley DB in proprietary applications. 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 Berkeley DB 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.

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 includes prebuilt binaries and libraries for a small number of platforms. The distribution does not include hard-copy documentation.

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 (page 489) 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, OpenLinux and others. See Building for UNIX/POSIX (page 289) for more information.

The Berkeley DB distribution includes support for QNX Neutrino. See Building for UNIX/POSIX (page 289) for more information.

The Berkeley DB distribution includes support for VxWorks. See Building for VxWorks 6.x (page 327) for more information.

The Berkeley DB distribution includes support for Windows/NT, Windows/2000 and Windows/XP, via the Microsoft Visual C++ 6.0 and .NET development environments. See Building Berkeley DB for Windows (page 311) for more information.

The Berkeley DB products

Oracle 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 Oracle, 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 from Oracle.

Berkeley DB Data Store

The Berkeley DB Data Store product is an embeddable, high-performance data store. It supports multiple concurrent threads of control (including multiple processes and multiple threads of control within a process) reading information managed by Berkeley DB. When updates are required, only a single thread of control may be using the database. 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 read-only applications or applications which can guarantee no more than one thread of control will ever update the database at a 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.

Chapter 2. Access Method Configuration

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 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() and DBC->get() methods with the DB_SET_RECNO flag specified, the **data** field of the key DBT 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 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 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 (page 36) for more information.

When appending new data items into Queue databases, record numbers 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, Berkeley DB will return the system error EFBIG. Record numbers do not wrap around when appending new data items into Recno databases.

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);
}
```

General access method configuration

There are a series of configuration tasks which are common to all access methods. They are described in the following sections.

Selecting a page size

The size of the pages used in the underlying database can be specified by calling the DB->set_pagesize() 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 (page 180) 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() 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 4KB to 8KB, 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 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 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()

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() method with the 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 DBC->put() method and one of the DB_AFTER, DB_BEFORE, DB_KEYFIRST or 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() method, as it always returns the first of the duplicate data items. Duplicate data items should be retrieved using a Berkeley DB cursor interface such as the DBC->get() method.

There is a 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 for more information. There is a 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 and 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() method with the DB_DUPSORT flag (in addition to the DB_DUP flag). In addition, a custom comparison function may be specified using the DB->set_dup_compare() method. If the 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 a flag that permits applications to disallow storing duplicate data items when the database has been configured for sorted duplicates, see DB_NODUPDATA for more information. Applications not wanting to permit duplicate duplicates in databases configured for sorted duplicates should begin using the 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() DB->put(), DBC->get() and DBC->put() 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() and DB->set_alloc() methods can be used to give Berkeley DB references to the application's allocation routines.

Btree access method specific configuration

There are a series of configuration tasks which you can perform when using the Btree access method. They are described in the following sections.

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() method.

Sort routines are passed pointers to keys as arguments. The keys are represented as DBT 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 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;
```

```
<para />
 /*
  * 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;
<para />
 /*
  * 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.

It is reasonable for a comparison function to not examine an entire key in some applications, which implies 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() and DBC->get() with the DB_SET flag), will modify the user-specified key by returning the actual key stored in the database.

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().

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 structures. The only fields the routines may examine in the DBT structures are **data** and **size** fields.

The prefix comparison function must return the number of bytes necessary to distinguish the two keys. If the keys are identical (equal and equal in length), the length should be returned. If the keys are equal up to the smaller of the two lengths, then the length of the smaller key plus 1 should be returned.

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() 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() method with the 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 flag to the DB->get() and DBC->get() methods. The following is an example of a routine that displays the data item for a Btree database created with the 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 flag to the DBC->get() 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);
}
```

Compression

The Btree access method supports the automatic compression of key/data pairs upon their insertion into the database. The key/data pairs are decompressed before they are returned to the application, making an application's interaction with a compressed database identical to that for a non-compressed database. To configure Berkeley DB for compression, call the DB->set_bt_compress() method and specify custom compression and decompression functions. If DB->set_bt_compress() is called with NULL compression and decompression functions, Berkeley DB will use its default compression functions.

Compression only works with the Btree access method, and then only so long as your database is not configured for unsorted duplicates.

The default compression function performs prefix compression on each key added to the database. This means that, for a key *n* bytes in length, the first *i* bytes that match the first *i* bytes of the previous key exactly are omitted and only the final *n-i* bytes are stored in the database. If the bytes of key being stored match the bytes of the previous key exactly, then the same prefix compression algorithm is applied to the data value being stored. To use Berkeley DB's default compression behavior, both the default compression and decompression functions must be used.

For example, to configure your database for default compression:

```
    DB *dbp = NULL;
    DB_ENV *envp = NULL;
    u_int32_t db_flags;
    const char *file_name = "mydb.db";
    int ret;

...

    /* Skipping environment open to shorten this example */
...

    /* Initialize the DB handle */
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        fprintf(stderr, "%s\n", db_strerror(ret));
        return (EXIT_FAILURE);
    }

    /* Turn on default data compression */
    dbp->set_bt_compress(dbp, NULL, NULL);

    /* Now open the database */
    db_flags = DB_CREATE;       /* Allow database creation */

    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    file_name,  /* File name */
                    NULL,       /* Logical db name */
                    DB_BTREE,   /* Database type (using btree) */
                    db_flags,   /* Open flags */
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database '%s' open failed",
            file_name);
        return (EXIT_FAILURE);
    }
```

Custom compression

An application wishing to perform it's own compression may supply a compression and decompression function which will be called instead of Berkeley DB's default functions. The compression function is passed five DBT structures:

- The key and data immediately preceeding the key/data pair that is being stored.
- The key and data being stored in the tree.
- The buffer where the compressed data should be written.

The total size of the buffer used to store the compressed data is identified in the DBT's `ulen` field. If the compressed data cannot fit in the buffer, the compression function should store the amount of space needed in DBT's `size` field and then return `DB_BUFFER_SMALL`. Berkeley DB will subsequently re-call the compression function with the required amount of space allocated in the compression data buffer.

Multiple compressed key/data pairs will likely be written to the same buffer and the compression function should take steps to ensure it does not overwrite data.

For example, the following code fragments illustrate the use of a custom compression routine. This code is actually a much simplified example of the default compression provided by Berkeley DB. It does simple prefix compression on the key part of the data.

```
int compress(DB *dbp, const DBT *prevKey, const DBT *prevData,
        const DBT *key, const DBT *data, DBT *dest)
{
    u_int8_t *dest_data_ptr;
    const u_int8_t *key_data, *prevKey_data;
    size_t len, prefix, suffix;

    key_data = (const u_int8_t*)key->data;
    prevKey_data = (const u_int8_t*)prevKey->data;
    len = key->size > prevKey->size ? prevKey->size : key->size;
    for (; len-- && *key_data == *prevKey_data; ++key_data,
                                                ++prevKey_data)
        continue;

    prefix = (size_t)(key_data - (u_int8_t*)key->data);
    suffix = key->size - prefix;

    /* Check that we have enough space in dest */
    dest->size = (u_int32_t)(__db_compress_count_int(prefix) +
        __db_compress_count_int(suffix) +
        __db_compress_count_int(data->size) + suffix + data->size);
    if (dest->size > dest->ulen)
        return (DB_BUFFER_SMALL);

    /* prefix length */
    dest_data_ptr = (u_int8_t*)dest->data;
    dest_data_ptr += __db_compress_int(dest_data_ptr, prefix);

    /* suffix length */
    dest_data_ptr += __db_compress_int(dest_data_ptr, suffix);

    /* data length */
    dest_data_ptr += __db_compress_int(dest_data_ptr, data->size);

    /* suffix */
    memcpy(dest_data_ptr, key_data, suffix);
```

```
    dest_data_ptr += suffix;

    /* data */
    memcpy(dest_data_ptr, data->data, data->size);

    return (0);
}
```

The corresponding decompression function is likewise passed five DBT structures:

- The key and data DBTs immediately preceding the decompressed key and data.
- The compressed data from the database.
- One to store the decompressed key and another one for the decompressed data.

Because the compression of `record X` relies upon `record X-1`, the decompression function can be called repeatedly to linearally decompress a set of records stored in the compressed buffer.

The total size of the buffer available to store the decompressed data is identified in the destination DBT's `ulen` field. If the decompressed data cannot fit in the buffer, the decompression function should store the amount of space needed in the destination DBT's `size` field and then return `DB_BUFFER_SMALL`. Berkeley DB will subsequently re-call the decompression function with the required amount of space allocated in the decompression data buffer.

For example, the decompression routine that corresponds to the example compression routine provided above is:

```
int decompress(DB *dbp, const DBT *prevKey, const DBT *prevData,
    DBT *compressed, DBT *destKey, DBT *destData)
{
    u_int8_t *comp_data, *dest_data;
    u_int32_t prefix, suffix, size;

    /* Unmarshal prefix, suffix and data length */
    comp_data = (u_int8_t*)compressed->data;
    size = __db_decompress_count_int(comp_data);
    if (size > compressed->size)
        return (EINVAL);
    comp_data += __db_decompress_int32(comp_data, &prefix);

    size += __db_decompress_count_int(comp_data);
    if (size > compressed->size)
        return (EINVAL);
    comp_data += __db_decompress_int32(comp_data, &suffix);

    size += __db_decompress_count_int(comp_data);
    if (size > compressed->size)
        return (EINVAL);
    comp_data += __db_decompress_int32(comp_data, &destData->size);
```

```
    /* Check destination lengths */
    destKey->size = prefix + suffix;
    if (destKey->size > destKey->ulen ||
            destData->size > destData->ulen)
        return (DB_BUFFER_SMALL);

    /* Write the prefix */
    if (prefix > prevKey->size)
        return (EINVAL);
    dest_data = (u_int8_t*)destKey->data;
    memcpy(dest_data, prevKey->data, prefix);
    dest_data += prefix;

    /* Write the suffix */
    size += suffix;
    if (size > compressed->size)
        return (EINVAL);
    memcpy(dest_data, comp_data, suffix);
    comp_data += suffix;

    /* Write the data */
    size += destData->size;
    if (size > compressed->size)
        return (EINVAL);
    memcpy(destData->data, comp_data, destData->size);
    comp_data += destData->size;

    /* Return bytes read */
    compressed->size =
        (u_int32_t)(comp_data - (u_int8_t*)compressed->data);
    return (0);
}
```

Programmer Notes

As you use compression with your databases, be aware of the following:

- Compression works by placing key/data pairs from a single database page into a single block of compressed data. This is true whether you use DB's default compression, or you write your own compression. Because all of key/data data is placed in a single block of memory, you cannot decompress data unless you have decompressed everything that came before it in the block. That is, you cannot decompress item *n* in the data block, unless you also decompress items *0* through *n-1*.
- If you increase the minimum number of key/data pairs placed on a Btree leaf page (using DB->set_bt_minkey()), you will decrease your seek times on a compressed database. However, this will also decrease the effectiveness of the compression.

- Compressed databases are fastest if bulk load is used to add data to them. See Retrieving and updating records in bulk (page 67) for information on using bulk load.

Hash access method specific configuration

There are a series of configuration tasks which you can perform when using the Hash access method. They are described in the following sections.

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() 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 the db_stat 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() 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 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() 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() method must also be called to set the page fill factor.

Queue and Recno access method specific configuration

There are a series of configuration tasks which you can perform when using the Queue and Recno access methods. They are described in the following sections.

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() method. 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() 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() 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 (page 35).

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() 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 (page 35).

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() 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() method. 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() or DB->sync() 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() 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() call. This can be accomplished by calling the DB->set_flags() method with the 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()), **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()), 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() method and the 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 handle can adjust each other's position in this way, however. If multiple DB 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 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 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 DBC->get() method with the 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 (page 230).

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 (page 230). 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 (page 230) return.

Chapter 3. Access Method Operations

Once a database handle has been created using db_create(), there are several standard access method operations. Each of these operations is performed using a method referred to by the returned handle. Generally, the database will be opened using DB->open(). If the database is from an old release of Berkeley DB, it may need to be upgraded to the current release before it is opened using DB->upgrade().

Once a database has been opened, records may be retrieved (DB->get()), stored (DB->put()), and deleted (DB->del()).

Additional operations supported by the database handle include statistics (DB->stat()), truncation (DB->truncate()), version upgrade (DB->upgrade()), verification and salvage (DB->verify()), flushing to a backing file (DB->sync()), and association of secondary indices (DB->associate()). Database handles are eventually closed using DB->close().

Database Operations	Description
db_create()	Create a database handle
DB->associate()	Associate a secondary index
DB->associate_foreign()	Associate a foreign index
DB->close()	Close a database
DB->compact()	Compact a database
DB->cursor()	Create a cursor
DB->del()	Delete items from a database
DB->err()	Error message
DB->existis()	Return if an item appears in a database
DB->fd()	Return a file descriptor from a database
DB->get()	Get items from a database
DB->get_byteswapped()	Return if the underlying database is in host order
DB->get_type()	Return the database type
DB->join()	Perform a database join on cursors
DB->key_range()	Return estimate of key location
DB->open()	Open a database
DB->put()	Store items into a database
DB->remove()	Remove a database
DB->rename()	Rename a database
DB->set_priority()	Set cache page priority
DB->stat()	Database statistics
DB->sync()	Flush a database to stable storage

Database Operations	Description
DB->truncate()	Empty a database
DB->upgrade()	Upgrade a database
DB->verify()	Verify/salvage a database
Database Configuration	
DB->set_alloc()	Set local space allocation functions
DB->set_cachesize()	Set the database cache size
DB->set_dup_compare()	Set a duplicate comparison function
DB->set_encrypt()	Set the database cryptographic key
DB->set_errcall()	Set error and informational message callback
DB->set_errfile()	Set error and informational message FILE
DB->set_errpfx()	Set error message prefix
DB->set_feedback()	Set feedback callback
DB->set_flags()	General database configuration
DB->set_lorder()	Set the database byte order
DB->set_pagesize()	Set the underlying database page size
Btree/Recno Configuration	
DB->set_append_recno()	Set record append callback
DB->set_bt_compare()	Set a Btree comparison function
DB->set_bt_minkey()	Set the minimum number of keys per Btree page
DB->set_bt_prefix()	Set a Btree prefix comparison function
DB->set_re_delim()	Set the variable-length record delimiter
DB->set_re_len()	Set the fixed-length record length
DB->set_re_pad()	Set the fixed-length record pad byte
DB->set_re_source()	Set the backing Recno text file
Hash Configuration	
DB->set_h_compare()	Set a Hash comparison function
DB->set_h_ffactor()	Set the Hash table density
DB->set_h_hash()	Set a hashing function
DB->set_h_nelem()	Set the Hash table size
Queue Configuration	
DB->set_q_extentsize()	Set Queue database extent size

Database open

The DB->open() method opens a database, and takes five arguments:

file

The name of the file to be opened.

database

An optional database name.

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.

mode

The permissions to give to any created file.

There are a few flags that you can set to customize open:

DB_CREATE

Create the underlying database and any necessary physical files.

DB_NOMMAP

Do not map this database into process memory.

DB_RDONLY

Treat the data base as read-only.

DB_THREAD

The returned handle is free-threaded, that is, it can be used simultaneously by multiple threads within the process.

DB_TRUNCATE

Physically truncate the underlying database file, discarding all databases it contained. Underlying filesystem primitives are used to implement this flag. For this reason it is only applicable to the physical file and cannot be used to discard individual databases from within physical files.

DB_UPGRADE

Upgrade the database format as necessary.

Opening multiple databases in a single file

Applications may create multiple databases within a single physical file. This is useful when the databases are both numerous and reasonably small, in order to avoid creating a large number of underlying files, or when it is desirable to include secondary index databases in the same file as the primary index database. Putting multiple databases in a single physical file is an administrative convenience and unlikely to affect database performance.

To open or create a file that will include more than a single database, specify a database name when calling the DB->open() method.

Physical files do not need to be comprised of a single type of database, and databases in a file may be of any mixture of types, except for Queue databases. Queue databases must be created one per file and cannot share a file with any other database type. There is no limit on the

number of databases that may be created in a single file other than the standard Berkeley DB file size and disk space limitations.

It is an error to attempt to open a second database in a file that was not initially created using a database name, that is, the file must initially be specified as capable of containing multiple databases for a second database to be created in it.

It is not an error to open a file that contains multiple databases without specifying a database name, however the database type should be specified as DB_UNKNOWN and the database must be opened read-only. The handle that is returned from such a call is a handle on a database whose key values are the names of the databases stored in the database file and whose data values are opaque objects. No keys or data values may be modified or stored using this database handle.

Configuring databases sharing a file

There are four pieces of configuration information which must be specified consistently for all databases in a file, rather than differing on a per-database basis. They are: byte order, checksum and encryption behavior, and page size. When creating additional databases in a file, any of these configuration values specified must be consistent with the existing databases in the file or an error will be returned.

Caching databases sharing a file

When storing multiple databases in a single physical file rather than in separate files, if any of the databases in a file is opened for update, all of the databases in the file must share a memory pool. In other words, they must be opened in the same database environment. This is so per-physical-file information common between the two databases is updated correctly.

Locking in databases based on sharing a file

If databases are in separate files (and access to each separate database is single-threaded), there is no reason to perform any locking of any kind, and the two databases may be read and written simultaneously. Further, there would be no requirement to create a shared database environment in which to open those two databases.

However, since multiple databases in a file exist in a single physical file, opening two databases in the same file simultaneously requires locking be enabled, unless all of the databases are read-only. As the locks for the two databases can only conflict during page allocation, this additional locking is unlikely to affect performance. The exception is when Berkeley DB Concurrent Data Store is configured; a single lock is used for all databases in the file when Berkeley DB Concurrent Data Store is configured, and a write to one database will block all accesses to all databases.

In summary, programmers writing applications that open multiple databases in a single file will almost certainly need to create a shared database environment in the application as well. For more information on database environments, see Database environment introduction (page 122)

Partitioning databases

You can improve concurrency on your database reads and writes by splitting access to a single database into multiple databases. This helps to avoid contention for internal database pages, as well as allowing you to spread your databases across multiple disks, which can help to improve disk I/O.

While you can manually do this by creating and using more than one database for your data, DB is capable of partitioning your database for you. When you use DB's built-in database partitioning feature, your access to your data is performed in exactly the same way as if you were only using one database; all the work of knowing which database to use to access a particular record is handled for you under the hood.

Only the BTree and Hash access methods are supported for partitioned databases.

You indicate that you want your database to be partitioned by calling DB->set_partition() before opening your database the first time. You can indicate the directory in which each partition is contained using the DB->set_partition_dirs() method.

Once you have partitioned a database, you cannot change your partitioning scheme.

There are two ways to indicate what key/data pairs should go on which partition. The first is by specifying an array of DBTs that indicate the minimum key value for a given partition. The second is by providing a callback that returns the number of the partition on which a specified key is placed.

Specifying partition keys

For simple cases, you can partition your database by providing an array of DBTs, each element of which provides the minimum key value to be placed on a partition. There must be one fewer elements in this array than you have partitions. The first element of the array indicates the minimum key value for the second partition in your database. Key values that are less than the first key value provided in this array are placed on the first partition (partition 0).

You can use partition keys only if you are using the Btree access method.

For example, suppose you had a database of fruit, and you want three partitions for your database. Then you need a DBT array of size two. The first element in this array indicates the minimum keys that should be placed on partition 1. The second element in this array indicates the minimum key value placed on partition 2. Keys that compare less than the first DBT in the array are placed on partition 0.

All comparisons are performed according to the lexicographic comparison used by your platform.

For example, suppose you want all fruits whose names begin with:

- 'a' - 'f' to go on partition 0
- 'g' - 'p' to go on partition 1

- 'q' - 'z' to go on partition 2.

Then you would accomplish this with the following code fragment:

The DB->set_partition() partition callback parameter must be NULL if you are using an array of DBTs to partition your database.

```
    DB *dbp = NULL;
    DB_ENV *envp = NULL;
    DBT partKeys[2];
    u_int32_t db_flags;
    const char *file_name = "mydb.db";
    int ret;

...

    /* Skipping environment open to shorten this example */
...

    /* Initialize the DB handle */
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        fprintf(stderr, "%s\n", db_strerror(ret));
        return (EXIT_FAILURE);
    }

    /* Setup the partition keys */
     memset(&partKeys[0], 0, sizeof(DBT));
     partKeys[0].data = "g";
     partKeys[0].size = sizeof("g") - 1;

     memset(&partKeys[1], 0, sizeof(DBT));
     partKeys[1].data = "q";
     partKeys[1].size = sizeof("q") - 1;

     dbp->set_partition(dbp, 3, partKeys, NULL);

    /* Now open the database */
    db_flags = DB_CREATE;       /* Allow database creation */

    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    file_name,  /* File name */
                    NULL,       /* Logical db name */
                    DB_BTREE,   /* Database type (using btree) */
                    db_flags,   /* Open flags */
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database '%s' open failed",
            file_name);
```

```
        return (EXIT_FAILURE);
    }
```

Partitioning callback

In some cases, a simple lexicographical comparison of key data will not sufficiently support a partitioning scheme. For those situations, you should write a partitioning function. This function accepts a pointer to the DB and the DBT, and it returns the number of the partition on which the key belongs.

Note that DB actually places the key on the partition calculated by:

```
returned_partition modulo number_of_partitions
```

Also, remember that if you use a partitioning function when you create your database, then you must use the same partitioning function every time you open that database in the future.

The following code fragment illustrates a partition callback:

```
u_int32_t db_partition_fn(DB *db, DBT *key) {
    char *key_data;
    u_int32_t ret_number;
    /* Obtain your key data, unpacking it as necessary
     * Here, we do the very simple thing just for illustrative purposes.
     */

    key_data = (char *)key->data;

    /* Here you would perform whatever comparison you require to determine
     * what partition the key belongs on. If you return either 0 or the
     * number of partitions in the database, the key is placed in the first
     * database partition. Else, it is placed on:
     *
     *       returned_number mod number_of_partitions
     */

    ret_number = 0;

    return ret_number;
}
```

You then cause your partition callback to be used by providing it to the DB->set_partition() method, as illustrated by the following code fragment.

The DB->set_partition() DBT array parameter must be `NULL` if you are using a partition call back to partition your database.

```
    DB *dbp = NULL;
    DB_ENV *envp = NULL;
    u_int32_t db_flags;
    const char *file_name = "mydb.db";
    int ret;
```

```
...

    /* Skipping environment open to shorten this example */
...

    /* Initialize the DB handle */
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        fprintf(stderr, "%s\n", db_strerror(ret));
        return (EXIT_FAILURE);
    }

     dbp->set_partition(dbp, 3, NULL, db_partition_fn);

    /* Now open the database */
    db_flags = DB_CREATE;       /* Allow database creation */

    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    file_name,  /* File name */
                    NULL,       /* Logical db name */
                    DB_BTREE,   /* Database type (using btree) */
                    db_flags,   /* Open flags */
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database '%s' open failed",
            file_name);
        return (EXIT_FAILURE);
    }
```

Placing partition files

When you partition a database, a database file is created on disk in the same way as if you were not partitioning the database. That is, this file uses the name you provide to the DB->open() `file` parameter.

However, DB then also creates a series of database files on disk, one for each partition that you want to use. These partition files share the same name as the database file name, but are also number sequentially. So if you create a database named `mydb.db`, and you create 3 partitions for it, then you will see the following database files on disk:

```
            mydb.db
            __dbp.mydb.db.000
            __dbp.mydb.db.001
            __dbp.mydb.db.002
```

All of the database's contents go into the numbered database files. You can cause these files to be placed in different directories (and, hence, different disk partitions or even disks) by using the DB->set_partition_dirs() method.

DB->set_partition_dirs() takes a NULL-terminated array of strings, each one of which should represent an existing filesystem directory.

If you are using an environment, the directories specified using DB->set_partition_dirs() must also be included in the environment list specified by DB_ENV->add_data_dir().

If you are not using an environment, then the the directories specified to DB->set_partition_dirs() can be either complete paths to currently existing directories, or paths relative to the application's current working directory.

Ideally, you will provide DB->set_partition_dirs() with an array that is the same size as the number of partitions you are creating for your database. Partition files are then placed according to the order that directories are contained in the array; partition 0 is placed in directory_array[0], partition 1 in directory_array[1], and so forth. However, if you provide an array of directories that is smaller than the number of database partitions, then the directories are used on a round-robin fashion.

You must call DB->set_partition_dirs() before you create your database, and before you open your database each time thereafter. The array provided to DB->set_partition_dirs() must not change after the database has been created.

Retrieving records

The DB->get() method retrieves records from the database. In general, DB->get() takes a key and returns the associated data from the database.

There are a few flags that you can set to customize retrieval:

DB_GET_BOTH

Search for a matching key and data item, that is, only return success if both the key and the data items match those stored in the database.

DB_RMW

Read-modify-write: acquire write locks instead of read locks during retrieval. This can enhance performance in threaded applications by reducing the chance of deadlock.

DB_SET_RECNO

If the underlying database is a Btree, and was configured so that it is possible to search it by logical record number, retrieve a specific record.

If the database has been configured to support duplicate records, DB->get() will always return the first data item in the duplicate set.

Storing records

The DB->put() method stores records into the database. In general, DB->put() takes a key and stores the associated data into the database.

There are a few flags that you can set to customize storage:

DB_APPEND

Simply append the data to the end of the database, treating the database much like a simple log. This flag is only valid for the Queue and Recno access methods.

DB_NOOVERWRITE

Only store the data item if the key does not already appear in the database.

If the database has been configured to support duplicate records, the DB->put() method will add the new data value at the end of the duplicate set. If the database supports sorted duplicates, the new data value is inserted at the correct sorted location.

Deleting records

The DB->del() method deletes records from the database. In general, DB->del() takes a key and deletes the data item associated with it from the database.

If the database has been configured to support duplicate records, the DB->del() method will remove all of the duplicate records. To remove individual duplicate records, you must use a Berkeley DB cursor interface.

Database statistics

The DB->stat() method returns a set of statistics about the underlying database, for example, the number of key/data pairs in the database, how the database was originally configured, and so on.

There is a flag you can set to avoid time-consuming operations:

DB_FAST_STAT

Return only information that can be acquired without traversing the entire database.

Database truncation

The DB->truncate() method empties a database of all records.

Database upgrade

When upgrading to a new release of Berkeley DB, it may be necessary to upgrade the on-disk format of already-created database files. **Berkeley DB database upgrades are done in place, and so are potentially destructive.** This means that if the system crashes during the upgrade procedure, or if the upgrade procedure runs out of disk space, the databases may be left in an inconsistent and unrecoverable state. To guard against failure, the procedures outlined in Upgrading Berkeley DB installations (page 332) should be carefully followed. If you are not performing catastrophic archival as part of your application upgrade process, you should at least copy your database to archival media, verify that your archival media is error-free and readable, and that copies of your backups are stored offsite!

The actual database upgrade is done using the DB->upgrade() method, or by dumping the database using the old version of the Berkeley DB software and reloading it using the current version.

After an upgrade, Berkeley DB applications must be recompiled to use the new Berkeley DB library before they can access an upgraded database. **There is no guarantee that applications compiled against previous releases of Berkeley DB will work correctly with an upgraded database format. Nor is there any guarantee that applications compiled against newer releases of Berkeley DB will work correctly with the previous database format.** We do guarantee that any archived database may be upgraded using a current Berkeley DB software release and the DB->upgrade() method, and there is no need to step-wise upgrade the database using intermediate releases of Berkeley DB. Sites should consider archiving appropriate copies of their application or application sources if they may need to access archived databases without first upgrading them.

Database verification and salvage

The DB->verify() method verifies that a file, and any databases it may contain, are uncorrupted. In addition, the method may optionally be called with a file stream argument to which all key/data pairs found in the database are output. There are two modes for finding key/data pairs to be output:

1. If the DB_SALVAGE flag is specified, the key/data pairs in the database are output. When run in this mode, the database is assumed to be largely uncorrupted. For example, the DB->verify() method will search for pages that are no longer linked into the database, and will output key/data pairs from such pages. However, key/data items that have been marked as deleted in the database will not be output, as the page structures are generally trusted in this mode.

2. If both the DB_SALVAGE and DB_AGGRESSIVE flags are specified, all possible key/data pairs are output. When run in this mode, the database is assumed to be seriously corrupted. For example, key/data pairs that have been deleted will re-appear in the output. In addition, because pages may have been subsequently reused and modified during normal database operations after the key/data pairs were deleted, it is not uncommon for apparently corrupted key/data pairs to be output in this mode, even when there is no corruption in the underlying database. The output will almost always have to be edited by hand or other means before the data is ready for reload into another database. We recommend that DB_SALVAGE be tried first, and DB_AGGRESSIVE only tried if the output from that first attempt is obviously missing data items or the data is sufficiently valuable that human review of the output is preferable to any kind of data loss.

Flushing the database cache

The DB->sync() method flushes all modified records from the database cache to disk.

It is important to understand that flushing cached information to disk only minimizes the window of opportunity for corrupted data, it does not eliminate the possibility.

While unlikely, it is possible for database corruption to happen if a system or application crash occurs while writing data to the database. To ensure that database corruption never occurs, applications must either:

- Use transactions and logging with automatic recovery.
- Use logging and application-specific recovery.
- Edit a copy of the database, and, once all applications using the database have successfully called DB->close(), use system operations (for example, the POSIX rename system call) to atomically replace the original database with the updated copy.

Database close

The DB->close() database handle closes the DB handle. By default, DB->close() also flushes all modified records from the database cache to disk.

There is one flag that you can set to customize DB->close():

DB_NOSYNC

Do not flush cached information to disk.

It is important to understand that flushing cached information to disk only minimizes the window of opportunity for corrupted data, it does not eliminate the possibility.

While unlikely, it is possible for database corruption to happen if a system or application crash occurs while writing data to the database. To ensure that database corruption never occurs, applications must either:

- Use transactions and logging with automatic recovery.
- Use logging and application-specific recovery.
- Edit a copy of the database, and, once all applications using the database have successfully called DB->close(), use system operations (for example, the POSIX rename system call) to atomically replace the original database with the updated copy.

Secondary indexes

A secondary index, put simply, is a way to efficiently access records in a database (the primary) by means of some piece of information other than the usual (primary) key. In Berkeley DB, this index is simply another database whose keys are these pieces of information (the secondary keys), and whose data are the primary keys. Secondary indexes can be created manually by the application; there is no disadvantage, other than complexity, to doing so. However, when the secondary key can be mechanically derived from the primary key and datum that it points to, as is frequently the case, Berkeley DB can automatically and transparently manage secondary indexes.

As an example of how secondary indexes might be used, consider a database containing a list of students at a college, each of whom has a unique student ID number. A typical database would use the student ID number as the key; however, one might also reasonably want to be

able to look up students by last name. To do this, one would construct a secondary index in which the secondary key was this last name.

In SQL, this would be done by executing something like the following:

```
CREATE TABLE students(student_id CHAR(4) NOT NULL,
 lastname CHAR(15), firstname CHAR(15), PRIMARY KEY(student_id));
CREATE INDEX lname ON students(lastname);
```

In Berkeley DB, this would work as follows (a Java API example is also available [second.javas]):

```
struct student_record {
 char student_id[4];
 char last_name[15];
 char first_name[15];
};

void
second()
{
 DB *dbp, *sdbp;
 int ret;

 /* Open/create primary */
 if ((ret = db_create(&dbp, dbenv, 0)) != 0)
  handle_error(ret);
 if ((ret = dbp->open(dbp, NULL,
     "students.db", NULL, DB_BTREE, DB_CREATE, 0600)) != 0)
  handle_error(ret);

 /*
  * Open/create secondary.  Note that it supports duplicate data
  * items, since last names might not be unique.
  */
 if ((ret = db_create(&sdbp, dbenv, 0)) != 0)
  handle_error(ret);
 if ((ret = sdbp->set_flags(sdbp, DB_DUP | DB_DUPSORT)) != 0)
  handle_error(ret);
 if ((ret = sdbp->open(sdbp, NULL,
     "lastname.db", NULL, DB_BTREE, DB_CREATE, 0600)) != 0)
  handle_error(ret);

 /* Associate the secondary with the primary. */
 if ((ret = dbp->associate(dbp, NULL, sdbp, getname, 0)) != 0)
  handle_error(ret);
}

/*
 * getname -- extracts a secondary key (the last name) from a primary
 *  key/data pair
```

```
 */
int
getname(secondary, pkey, pdata, skey)
 DB *secondary;
 const DBT *pkey, *pdata;
 DBT *skey;
{
 /*
  * Since the secondary key is a simple structure member of the
  * record, we don't have to do anything fancy to return it.  If
  * we have composite keys that need to be constructed from the
  * record, rather than simply pointing into it, then the user's
  * function might need to allocate space and copy data.  In
  * this case, the DB_DBT_APPMALLOC flag should be set in the
  * secondary key DBT.
  */
 memset(skey, 0, sizeof(DBT));
 skey->data = ((struct student_record *)pdata->data)->last_name;
 skey->size = sizeof((struct student_record *)pdata->data)->last_name;
 return (0);
}
```

From the application's perspective, putting things into the database works exactly as it does without a secondary index; one can simply insert records into the primary database. In SQL one would do the following:

```
INSERT INTO student
    VALUES ("WC42", "Churchill      ", "Winston        ");
```

and in Berkeley DB, one does:

```
struct student_record s;
DBT data, key;

memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));
memset(&s, 0, sizeof(struct student_record));
key.data = "WC42";
key.size = 4;
memcpy(&s.student_id, "WC42", sizeof(s.student_id));
memcpy(&s.last_name, "Churchill      ", sizeof(s.last_name));
memcpy(&s.first_name, "Winston        ", sizeof(s.first_name));
data.data = &s;
data.size = sizeof(s);
if ((ret = dbp->put(dbp, txn, &key, &data, 0)) != 0)
 handle_error(ret);
```

Internally, a record with secondary key "Churchill" is inserted into the secondary database (in addition to the insertion of "WC42" into the primary, of course).

Deletes are similar. The SQL clause:

```
DELETE FROM student WHERE (student_id = "WC42");
```

looks like:

```
DBT key;

memset(&key, 0, sizeof(DBT));
key.data = "WC42";
key.size = 4;
if ((ret = dbp->del(dbp, txn, &key, 0)) != 0)
 handle_error(ret);
```

Deletes can also be performed on the secondary index directly; a delete done this way will delete the "real" record in the primary as well. If the secondary supports duplicates and there are duplicate occurrences of the secondary key, then all records with that secondary key are removed from both the secondary index and the primary database. In SQL:

```
DELETE FROM lname WHERE (lastname = "Churchill      ");
```

In Berkeley DB:

```
DBT skey;

memset(&skey, 0, sizeof(DBT));
skey.data = "Churchill      ";
skey.size = 15;
if ((ret = sdbp->del(sdbp, txn, &skey, 0)) != 0)
 handle_error(ret);
```

Gets on a secondary automatically return the primary datum. If DB->pget() or DBC->pget() is used in lieu of DB->get() or DBC->get(), the primary key is returned as well. Thus, the equivalent of:

```
SELECT * from lname WHERE (lastname = "Churchill      ");
```

would be:

```
DBT data, pkey, skey;
<para />
memset(&skey, 0, sizeof(DBT));
memset(&pkey, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));
skey.data = "Churchill      ";
skey.size = 15;
if ((ret = sdbp->pget(sdbp, txn, &skey, &pkey, &data, 0)) != 0)
 handle_error(ret);
/*
 * Now pkey contains "WC42" and data contains Winston's record.
 */
```

To create a secondary index to a Berkeley DB database, open the database that is to become a secondary index normally, then pass it as the "secondary" argument to the DB->associate() method for some primary database.

After a DB->associate() call is made, the secondary indexes become alternate interfaces to the primary database. All updates to the primary will be automatically reflected in each secondary index that has been associated with it. All get operations using the DB->get() or DBC->get() methods on the secondary index return the primary datum associated with the specified (or otherwise current, in the case of cursor operations) secondary key. The DB->pget() and DBC->pget() methods also become usable; these behave just like DB->get() and DBC->get(), but return the primary key in addition to the primary datum, for those applications that need it as well.

Cursor get operations on a secondary index perform as expected; although the data returned will by default be those of the primary database, a position in the secondary index is maintained normally, and records will appear in the order determined by the secondary key and the comparison function or other structure of the secondary database.

Delete operations on a secondary index delete the item from the primary database and all relevant secondaries, including the current one.

Put operations of any kind are forbidden on secondary indexes, as there is no way to specify a primary key for a newly put item. Instead, the application should use the DB->put() or DBC->put() methods on the primary database.

Any number of secondary indexes may be associated with a given primary database, up to limitations on available memory and the number of open file descriptors.

Note that although Berkeley DB guarantees that updates made using any DB handle with an associated secondary will be reflected in the that secondary, associating each primary handle with all the appropriate secondaries is the responsibility of the application and is not enforced by Berkeley DB. It is generally unsafe, but not forbidden by Berkeley DB, to modify a database that has secondary indexes without having those indexes open and associated. Similarly, it is generally unsafe, but not forbidden, to modify a secondary index directly. Applications that violate these rules face the possibility of outdated or incorrect results if the secondary indexes are later used.

If a secondary index becomes outdated for any reason, it should be discarded using the DB->remove() method and a new one created using the DB->associate() method. If a secondary index is no longer needed, all of its handles should be closed using the DB->close() method, and then the database should be removed using a new database handle and the DB->remove() method.

Closing a primary database handle automatically dis-associates all secondary database handles associated with it.

Foreign key indexes

Foreign keys are used to ensure a level of consistency between two different databases in terms of the keys that the databases use. In a foreign key relationship, one database is the

constrained database. This database is actually a secondary database which is associated with a primary database. The other database in this relationship is the *foreign key* database. Once this relationship has been established between a constrained database and a foreign key database, then:

1. Key/data items cannot be added to the constrained database unless that same key already exists in the foreign key database.

2. A key/data pair cannot be deleted from the foreign key database unless some action is also taken to keep the constrained database consistent with the foreign key database.

Because the constrained database is a secondary database, by ensuring it's consistency with a foreign key database you are actually ensuring that a primary database (the one to which the secondary database is associated) is consistent with the foreign key database.

Deletions of keys in the foreign key database affect the constrained database in one of three ways, as specified by the application:

- `Abort`

  The deletion of a record from the foreign database will not proceed if that key exists in the constrained primary database. Transactions must be used to prevent the aborted delete from corrupting either of the databases.

- `Cascade`

  The deletion of a record from the foreign database will also cause any records in the constrained primary database that use that key to also be automatically deleted.

- `Nullify`

  The deletion of a record from the foreign database will cause a user specified callback function to be called, in order to alter or nullify any records using that key in the constrained primary database.

Note that it is possible to delete a key from the constrained database, but not from the foreign key database. For this reason, if you want the keys used in both databases to be 100% accurate, then you will have to write code to ensure that when a key is removed from the constrained database, it is also removed from the foreign key database.

As an example of how foreign key indexes might be used, consider a database of customer information and a database of order information. A typical customer database would use a customer ID as the key and those keys would also appear in the order database. To ensure an order is not booked for a non-existent customer, the customer database can be associated with the order database as a foreign index.

In order to do this, you create a secondary index of the order database, which uses customer IDs as the key for its key/data pairs. This secondary index is, then, the constrained database. But because the secondary index is constrained, so too is the order database because the contents of the secondary index are programmatically tied to the contents of the order database.

The customer database, then, is the foreign key database. It is associated to the order database's secondary index using the DB->associate_foreign() method. In this way, an order cannot be added to the order database unless the customer ID already exists in the customer database.

Note that this relationship can also be configured to delete any outstanding orders for a customer when that customer is deleted from the customer database.

In SQL, this would be done by executing something like the following:

```
CREATE TABLE customers(cust_id CHAR(4) NOT NULL,
       lastname CHAR(15), firstname CHAR(15), PRIMARY KEY(cust_id));
CREATE TABLE orders(order_id CHAR(4) NOT NULL, order_num int NOT NULL,
       cust_id CHAR(4), PRIMARY KEY (order_id),
       FOREIGN KEY (cust_id) REFERENCES customers(cust_id)
       ON DELETE CASCADE);
```

In Berkeley DB, this would work as follows:

```
struct customer {
       char cust_id[4];
       char last_name[15];
       char first_name[15];
};
struct order {
       char order_id[4];
       int order_number;
       char cust_id[4];
};

void
foreign()
{
       DB *dbp, *sdbp, *fdbp;
       int ret;

       /* Open/create order database */
       if ((ret = db_create(&dbp, dbenv, 0)) != 0)
              handle_error(ret);
       if ((ret = dbp->open(dbp, NULL,
           "orders.db", NULL, DB_BTREE, DB_CREATE, 0600)) != 0)
              handle_error(ret);

       /*
        * Open/create secondary index on customer id.  Note that it
        * supports duplicates because a customer may have multiple
        * orders.
        */
       if ((ret = db_create(&sdbp, dbenv, 0)) != 0)
              handle_error(ret);
       if ((ret = sdbp->set_flags(sdbp, DB_DUP | DB_DUPSORT)) != 0)
```

```
                handle_error(ret);
        if ((ret = sdbp->open(sdbp, NULL, "orders_cust_ids.db",
            NULL, DB_BTREE, DB_CREATE, 0600)) != 0)
                handle_error(ret);

        /* Associate the secondary with the primary. */
        if ((ret = dbp->associate(dbp, NULL, sdbp, getcustid, 0)) != 0)
                handle_error(ret);

        /* Open/create customer database */
        if ((ret = db_create(&fdbp, dbenv, 0)) != 0)
                handle_error(ret);
        if ((ret = fdbp->open(fdbp, NULL,
            "customers.db", NULL, DB_BTREE, DB_CREATE, 0600)) != 0)
                handle_error(ret);

        /* Associate the foreign with the secondary. */
        if ((ret = fdbp->associate_foreign(
            fdbp, sdbp, NULL, DB_FOREIGN_CASCADE)) != 0)
                handle_error(ret);

}

/*
* getcustid -- extracts a secondary key (the customer id) from a primary
*     key/data pair
*/
int
getname(secondary, pkey, pdata, skey)
        DB *secondary;
        const DBT *pkey, *pdata;
        DBT *skey;
{
        /*
         * Since the secondary key is a simple structure member of the
         * record, we don't have to do anything fancy to return it.  If
         * we have composite keys that need to be constructed from the
         * record, rather than simply pointing into it, then the user's
         * function might need to allocate space and copy data.  In
         * this case, the DB_DBT_APPMALLOC flag should be set in the
         * secondary key DBT.
         */
        memset(skey, 0, sizeof(DBT));
        skey->data = ((struct order *)pdata->data)->cust_id;
        skey->size = 4;
        return (0);
}
```

Cursor operations

A database cursor refers to a single key/data pair in the database. It supports traversal of the database and is the only way to access individual duplicate data items. Cursors are used for operating on collections of records, for iterating over a database, and for saving handles to individual records, so that they can be modified after they have been read.

The DB->cursor() method opens a cursor into a database. Upon return the cursor is uninitialized, cursor positioning occurs as part of the first cursor operation.

Once a database cursor has been opened, records may be retrieved (DBC->get()), stored (DBC->put()), and deleted (DBC->del()).

Additional operations supported by the cursor handle include duplication (DBC->dup()), equality join (DB->join()), and a count of duplicate data items (DBC->count()). Cursors are eventually closed using DBC->close().

Database Cursors and Related Methods	Description
DB->cursor()	Create a cursor
DBC->close()	Close a cursor
DBC->count()	Return count of duplicates
DBC->del()	Delete by cursor
DBC->dup()	Duplicate a cursor
DBC->get()	Retrieve by cursor
DBC->put()	Store by cursor
DBC->set_priority()	Set the cursor's cache priority

Retrieving records with a cursor

The DBC->get() method retrieves records from the database using a cursor. The DBC->get() method takes a flag which controls how the cursor is positioned within the database and returns the key/data item associated with that positioning. Similar to DB->get(), DBC->get() may also take a supplied key and retrieve the data associated with that key from the database. There are several flags that you can set to customize retrieval.

Cursor position flags

DB_FIRST, DB_LAST

Return the first (last) record in the database.

DB_NEXT, DB_PREV

Return the next (previous) record in the database.

DB_NEXT_DUP

Return the next record in the database, if it is a duplicate data item for the current key.

DB_NEXT_NODUP, DB_PREV_NODUP

Return the next (previous) record in the database that is not a duplicate data item for the current key.

DB_CURRENT

Return the record from the database to which the cursor currently refers.

Retrieving specific key/data pairs

DB_SET

Return the record from the database that matches the supplied key. In the case of duplicates the first duplicate is returned and the cursor is positioned at the beginning of the duplicate list. The user can then traverse the duplicate entries for the key.

DB_SET_RANGE

Return the smallest record in the database greater than or equal to the supplied key. This functionality permits partial key matches and range searches in the Btree access method.

DB_GET_BOTH

Return the record from the database that matches both the supplied key and data items. This is particularly useful when there are large numbers of duplicate records for a key, as it allows the cursor to easily be positioned at the correct place for traversal of some part of a large set of duplicate records.

DB_GET_BOTH_RANGE

Return the smallest record in the database greater than or equal to the supplied key and data items.

Retrieving based on record numbers

DB_SET_RECNO

If the underlying database is a Btree, and was configured so that it is possible to search it by logical record number, retrieve a specific record based on a record number argument.

DB_GET_RECNO

If the underlying database is a Btree, and was configured so that it is possible to search it by logical record number, return the record number for the record to which the cursor refers.

Special-purpose flags

DB_CONSUME

Read-and-delete: the first record (the head) of the queue is returned and deleted. The underlying database must be a Queue.

DB_RMW

Read-modify-write: acquire write locks instead of read locks during retrieval. This can enhance performance in threaded applications by reducing the chance of deadlock.

In all cases, the cursor is repositioned by a DBC->get() operation to point to the newly-returned key/data pair in the database.

The following is a code example showing a cursor walking through a database and displaying the records it contains to the standard output:

```
int
display(database)
 char *database;
{
 DB *dbp;
 DBC *dbcp;
 DBT key, data;
 int close_db, close_dbc, ret;

 close_db = close_dbc = 0;

 /* Open the database. */
 if ((ret = db_create(&dbp, NULL, 0)) != 0) {
  fprintf(stderr,
      "%s: db_create: %s\n", progname, db_strerror(ret));
  return (1);
 }
 close_db = 1;

 /* Turn on additional error output. */
 dbp->set_errfile(dbp, stderr);
 dbp->set_errpfx(dbp, progname);

 /* Open the database. */
 if ((ret = dbp->open(dbp, NULL, database, NULL,
            DB_UNKNOWN, DB_RDONLY, 0)) != 0) {
  dbp->err(dbp, ret, "%s: DB->open", database);
  goto err;
 }

 /* Acquire a cursor for the database. */
 if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {
  dbp->err(dbp, ret, "DB->cursor");
  goto err;
 }
 close_dbc = 1;

 /* Initialize the key/data return 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("%.*s : %.*s\n",
      (int)key.size, (char *)key.data,
      (int)data.size, (char *)data.data);
 if (ret != DB_NOTFOUND) {
  dbp->err(dbp, ret, "DBcursor->get");
  goto err;
 }

err: if (close_dbc && (ret = dbcp->c_close(dbcp)) != 0)
  dbp->err(dbp, ret, "DBcursor->close");
 if (close_db && (ret = dbp->close(dbp, 0)) != 0)
  fprintf(stderr,
      "%s: DB->close: %s\n", progname, db_strerror(ret));
 return (0);
}
```

Storing records with a cursor

The DBC->put() method stores records into the database using a cursor. In general, DBC->put() takes a key and inserts the associated data into the database, at a location controlled by a specified flag.

There are several flags that you can set to customize storage:

DB_AFTER

Create a new record, immediately after the record to which the cursor refers.

DB_BEFORE

Create a new record, immediately before the record to which the cursor refers.

DB_CURRENT

Replace the data part of the record to which the cursor refers.

DB_KEYFIRST

Create a new record as the first of the duplicate records for the supplied key.

DB_KEYLAST

Create a new record, as the last of the duplicate records for the supplied key.

In all cases, the cursor is repositioned by a DBC->put() operation to point to the newly inserted key/data pair in the database.

The following is a code example showing a cursor storing two data items in a database that supports duplicate data items:

```
int
store(dbp)
 DB *dbp;
{
 DBC *dbcp;
 DBT key, data;
 int ret;
```

```
 /*
  * The DB handle for a Btree database supporting duplicate data
  * items is the argument; acquire a cursor for the database.
  */
 if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {
  dbp->err(dbp, ret, "DB->cursor");
  goto err;
 }

 /* Initialize the key. */
 memset(&key, 0, sizeof(key));
 key.data = "new key";
 key.size = strlen(key.data) + 1;

 /* Initialize the data to be the first of two duplicate records. */
 memset(&data, 0, sizeof(data));
 data.data = "new key's data: entry #1";
 data.size = strlen(data.data) + 1;

 /* Store the first of the two duplicate records. */
 if ((ret = dbcp->c_put(dbcp, &key, &data, DB_KEYFIRST)) != 0)
  dbp->err(dbp, ret, "DB->cursor");

 /* Initialize the data to be the second of two duplicate records. */
 data.data = "new key's data: entry #2";
 data.size = strlen(data.data) + 1;

 /*
  * Store the second of the two duplicate records.  No duplicate
  * record sort function has been specified, so we explicitly
  * store the record as the last of the duplicate set.
  */
 if ((ret = dbcp->c_put(dbcp, &key, &data, DB_KEYLAST)) != 0)
  dbp->err(dbp, ret, "DB->cursor");

err: if ((ret = dbcp->c_close(dbcp)) != 0)
   dbp->err(dbp, ret, "DBcursor->close");

 return (0);
}
```

Deleting records with a cursor

The DBC->del() method deletes records from the database using a cursor. The DBC->del() method deletes the record to which the cursor currently refers. In all cases, the cursor position is unchanged after a delete.

Duplicating a cursor

Once a cursor has been initialized (for example, by a call to DBC->get()), it can be thought of as identifying a particular location in a database. The DBC->dup() method permits an application to create a new cursor that has the same locking and transactional information as the cursor from which it is copied, and which optionally refers to the same position in the database.

In order to maintain a cursor position when an application is using locking, locks are maintained on behalf of the cursor until the cursor is closed. In cases when an application is using locking without transactions, cursor duplication is often required to avoid self-deadlocks. For further details, refer to Berkeley DB Transactional Data Store locking conventions (page 253).

Equality Join

Berkeley DB supports "equality" (also known as "natural"), joins on secondary indices. An equality join is a method of retrieving data from a primary database using criteria stored in a set of secondary indices. It requires the data be organized as a primary database which contains the primary key and primary data field, and a set of secondary indices. Each of the secondary indices is indexed by a different secondary key, and, for each key in a secondary index, there is a set of duplicate data items that match the primary keys in the primary database.

For example, let's assume the need for an application that will return the names of stores in which one can buy fruit of a given color. We would first construct a primary database that lists types of fruit as the key item, and the store where you can buy them as the data item:

Primary key:	Primary data:
apple	Convenience Store
blueberry	Farmer's Market
peach	Shopway
pear	Farmer's Market
raspberry	Shopway
strawberry	Farmer's Market

We would then create a secondary index with the key **color**, and, as the data items, the names of fruits of different colors.

Secondary key:	Secondary data:
blue	blueberry
red	apple
red	raspberry
red	strawberry
yellow	peach
yellow	pear

This secondary index would allow an application to look up a color, and then use the data items to look up the stores where the colored fruit could be purchased. For example, by first looking up **blue**, the data item **blueberry** could be used as the lookup key in the primary database, returning **Farmer's Market**.

Your data must be organized in the following manner in order to use the DB->join() method:

1. The actual data should be stored in the database represented by the DB object used to invoke this method. Generally, this DB object is called the *primary*.

2. Secondary indices should be stored in separate databases, whose keys are the values of the secondary indices and whose data items are the primary keys corresponding to the records having the designated secondary key value. It is acceptable (and expected) that there may be duplicate entries in the secondary indices.
   These duplicate entries should be sorted for performance reasons, although it is not required. For more information see the DB_DUPSORT flag to the DB->set_flags() method.

What the DB->join() method does is review a list of secondary keys, and, when it finds a data item that appears as a data item for all of the secondary keys, it uses that data item as a lookup into the primary database, and returns the associated data item.

If there were another secondary index that had as its key the **cost** of the fruit, a similar lookup could be done on stores where inexpensive fruit could be purchased:

Secondary key:	Secondary data:
expensive	blueberry
expensive	peach
expensive	pear
expensive	strawberry
inexpensive	apple
inexpensive	pear
inexpensive	raspberry

The DB->join() method provides equality join functionality. While not strictly cursor functionality, in that it is not a method off a cursor handle, it is more closely related to the cursor operations than to the standard DB operations.

It is also possible to do lookups based on multiple criteria in a single operation. For example, it is possible to look up fruits that are both red and expensive in a single operation. If the same fruit appeared as a data item in both the color and expense indices, then that fruit name would be used as the key for retrieval from the primary index, and would then return the store where expensive, red fruit could be purchased.

Example

Consider the following three databases:

personnel

- key = SSN
- data = record containing name, address, phone number, job title

lastname

- key = lastname
- data = SSN

jobs

- key = job title
- data = SSN

Consider the following query:

```
Return the personnel records of all people named smith with the job
title manager.
```

This query finds are all the records in the primary database (personnel) for whom the criteria **lastname=smith and job title=manager** is true.

Assume that all databases have been properly opened and have the handles: pers_db, name_db, job_db. We also assume that we have an active transaction to which the handle txn refers.

```
DBC *name_curs, *job_curs, *join_curs;
DBC *carray[3];
DBT key, data;
int ret, tret;

name_curs = NULL;
job_curs = NULL;
memset(&key, 0, sizeof(key));
memset(&data, 0, sizeof(data));

if ((ret =
    name_db->cursor(name_db, txn, &name_curs, 0)) != 0)
 goto err;
key.data = "smith";
key.size = sizeof("smith");
if ((ret =
    name_curs->c_get(name_curs, &key, &data, DB_SET)) != 0)
 goto err;

if ((ret = job_db->cursor(job_db, txn, &job_curs, 0)) != 0)
 goto err;
key.data = "manager";
key.size = sizeof("manager");
```

```
if ((ret =
    job_curs->c_get(job_curs, &key, &data, DB_SET)) != 0)
 goto err;

carray[0] = name_curs;
carray[1] = job_curs;
carray[2] = NULL;

if ((ret =
    pers_db->join(pers_db, carray, &join_curs, 0)) != 0)
 goto err;
while ((ret =
    join_curs->c_get(join_curs, &key, &data, 0)) == 0) {
 /* Process record returned in key/data. */
}

/*
 * If we exited the loop because we ran out of records,
 * then it has completed successfully.
 */
if (ret == DB_NOTFOUND)
 ret = 0;

err:
if (join_curs != NULL &&
    (tret = join_curs->c_close(join_curs)) != 0 && ret == 0)
 ret = tret;
if (name_curs != NULL &&
    (tret = name_curs->c_close(name_curs)) != 0 && ret == 0)
 ret = tret;
if (job_curs != NULL &&
    (tret = job_curs->c_close(job_curs)) != 0 && ret == 0)
 ret = tret;

return (ret);
```

The name cursor is positioned at the beginning of the duplicate list for **smith** and the job cursor is placed at the beginning of the duplicate list for **manager**. The join cursor is returned from the join method. This code then loops over the join cursor getting the personnel records of each one until there are no more.

Data item count

Once a cursor has been initialized to refer to a particular key in the database, it can be used to determine the number of data items that are stored for any particular key. The DBC->count() method returns this number of data items. The returned value is always one, unless the database supports duplicate data items, in which case it may be any number of items.

Cursor close

The DBC->close() method closes the DBC cursor, after which the cursor may no longer be used. Although cursors are implicitly closed when the database they point to are closed, it is good programming practice to explicitly close cursors. In addition, in transactional systems, cursors may not exist outside of a transaction and so must be explicitly closed.

Chapter 4. Access Method Wrapup

Data alignment

The Berkeley DB access methods provide no guarantees about byte alignment for returned key/data pairs, or callback functions which take DBT references as arguments, and applications are responsible for arranging any necessary alignment. The DB_DBT_MALLOC, DB_DBT_REALLOC, and DB_DBT_USERMEM flags may be used to store returned items in memory of arbitrary alignment.

Retrieving and updating records in bulk

When retrieving or modifying large numbers of records, the number of method calls can often dominate performance. Berkeley DB offers bulk get, put and delete interfaces which can significantly increase performance for some applications.

Bulk retrieval

To retrieve records in bulk, an application buffer must be specified to the DB->get() or DBC->get() methods. This is done in the C API by setting the **data** and **ulen** fields of the **data** DBT to reference an application buffer, and the **flags** field of that structure to DB_DBT_USERMEM. In the Berkeley DB C++ and Java APIs, the actions are similar, although there are API-specific methods to set the DBT values. Then, the DB_MULTIPLE or DB_MULTIPLE_KEY flags are specified to the DB->get() or DBC->get() methods, which cause multiple records to be returned in the specified buffer.

The difference between DB_MULTIPLE and DB_MULTIPLE_KEY is as follows: DB_MULTIPLE returns multiple data items for a single key. For example, the DB_MULTIPLE flag would be used to retrieve all of the duplicate data items for a single key in a single call. The DB_MULTIPLE_KEY flag is used to retrieve multiple key/data pairs, where each returned key may or may not have duplicate data items.

Once the DB->get() or DBC->get() method has returned, the application will walk through the buffer handling the returned records. This is implemented for the C and C++ APIs using four macros: DB_MULTIPLE_INIT, DB_MULTIPLE_NEXT, DB_MULTIPLE_KEY_NEXT, and DB_MULTIPLE_RECNO_NEXT. For the Java API, this is implemented as three iterator classes: MultipleDataEntry [../java/com/sleepycat/db/MultipleDataEntry.html], MultipleKeyDataEntry [../java/com/sleepycat/db/MultipleKeyDataEntry.html], and MultipleRecnoDataEntry [../java/com/sleepycat/db/MultipleRecnoDataEntry.html].

The DB_MULTIPLE_INIT macro is always called first. It initializes a local application variable and the **data** DBT for stepping through the set of returned records. Then, the application calls one of the remaining three macros: DB_MULTIPLE_NEXT, DB_MULTIPLE_KEY_NEXT, and DB_MULTIPLE_RECNO_NEXT.

If the DB_MULTIPLE flag was specified to the DB->get() or DBC->get() method, the application will always call the DB_MULTIPLE_NEXT macro. If the DB_MULTIPLE_KEY flag was specified to the DB->get() or DBC->get() method, and the underlying database is a Btree or Hash database,

the application will always call the DB_MULTIPLE_KEY_NEXT macro. If the DB_MULTIPLE_KEY flag was specified to the DB->get() or DBC->get() method, and the underlying database is a Queue or Recno database, the application will always call the DB_MULTIPLE_RECNO_NEXT macro. The DB_MULTIPLE_NEXT, DB_MULTIPLE_KEY_NEXT, and DB_MULTIPLE_RECNO_NEXT macros are called repeatedly, until the end of the returned records is reached. The end of the returned records is detected by the application's local pointer variable being set to NULL.

The following is an example of a routine that displays the contents of a Btree database using the bulk return interfaces.

```
int
rec_display(dbp)
 DB *dbp;
{
 DBC *dbcp;
 DBT key, data;
 size_t retklen, retdlen;
 char *retkey, *retdata;
 int ret, t_ret;
 void *p;

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

 /* Review the database in 5MB chunks. */
#define BUFFER_LENGTH (5 * 1024 * 1024)
 if ((data.data = malloc(BUFFER_LENGTH)) == NULL)
  return (errno);
 data.ulen = BUFFER_LENGTH;
 data.flags = DB_DBT_USERMEM;

 /* Acquire a cursor for the database. */
 if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {
  dbp->err(dbp, ret, "DB->cursor");
  free(data.data);
  return (ret);
 }

 for (;;) {
  /*
         * Acquire the next set of key/data pairs.  This code
         * does not handle single key/data pairs that won't fit
         * in a BUFFER_LENGTH size buffer, instead returning
         * DB_BUFFER_SMALL to our caller.
   */
  if ((ret = dbcp->c_get(dbcp,
      &key, &data, DB_MULTIPLE_KEY | DB_NEXT)) != 0) {
   if (ret != DB_NOTFOUND)
    dbp->err(dbp, ret, "DBcursor->c_get");
```

```
    break;
   }

   for (DB_MULTIPLE_INIT(p, &data);;) {
    DB_MULTIPLE_KEY_NEXT(p,
        &data, retkey, retklen, retdata, retdlen);
    if (p == NULL)
     break;
    printf("key: %.*s, data: %.*s\n",
        (int)retklen, retkey, (int)retdlen, retdata);
   }
  }

  if ((t_ret = dbcp->c_close(dbcp)) != 0) {
   dbp->err(dbp, ret, "DBcursor->close");
   if (ret == 0)
    ret = t_ret;
  }

  free(data.data);

  return (ret);
 }
```

Bulk updates

To put records in bulk with the btree or hash access methods, construct bulk buffers in the **key** and **data** DBT using DB_MULTIPLE_WRITE_INIT and DB_MULTIPLE_WRITE_NEXT. To put records in bulk with the recno or queue access methods, construct bulk buffers in the **data** DBT as before, but construct the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT and DB_MULTIPLE_RECNO_WRITE_NEXT with a data size of zero;. In both cases, set the DB_MULTIPLE flag to DB->put().

Alternatively, for btree and hash access methods, construct a single bulk buffer in the **key** DBT using DB_MULTIPLE_WRITE_INIT and DB_MULTIPLE_KEY_WRITE_NEXT. For recno and queue access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT and DB_MULTIPLE_RECNO_WRITE_NEXT. In both cases, set the DB_MULTIPLE_KEY flag to DB->put().

A successful bulk operation is logically equivalent to a loop through each key/data pair, performing a DB->put() for each one.

Bulk deletes

To delete all records with a specified set of keys with the btree or hash access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_WRITE_INIT and DB_MULTIPLE_WRITE_NEXT. To delete a set of records with the recno or queue access methods, construct the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT and DB_MULTIPLE_RECNO_WRITE_NEXT with a data size of zero. In both cases, set the DB_MULTIPLE

flag to DB->del(). This is equivalent to calling DB->del() for each key in the bulk buffer. In particular, if the database supports duplicates, all records with the matching key are deleted.

Alternatively, to delete a specific set of key/data pairs, which may be items within a set of duplicates, there are also two cases depending on whether the access method uses record numbers for keys. For btree and hash access methods, construct a single bulk buffer in the **key** DBT using DB_MULTIPLE_WRITE_INIT and DB_MULTIPLE_KEY_WRITE_NEXT. For recno and queue access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT and DB_MULTIPLE_RECNO_WRITE_NEXT. In both cases, set the DB_MULTIPLE_KEY flag to DB->del().

A successful bulk operation is logically equivalent to a loop through each key/data pair, performing a DB->del() for each one.

Partial record storage and retrieval

It is possible to both store and retrieve parts of data items in all Berkeley DB access methods. This is done by setting the DB_DBT_PARTIAL flag DBT structure passed to the Berkeley DB method.

The DB_DBT_PARTIAL flag is based on the values of two fields of the DBT structure: **dlen** and **doff**. The value of **dlen** is the number of bytes of the record in which the application is interested. The value of **doff** is the offset from the beginning of the data item where those bytes start.

For example, if the data item were **ABCDEFGHIJKL**, a **doff** value of 3 would indicate that the bytes of interest started at **D**, and a **dlen** value of 4 would indicate that the bytes of interest were **DEFG**.

When retrieving a data item from a database, the **dlen** bytes starting **doff** bytes from the beginning of the record are returned, as if they comprised the entire record. If any or all of the specified bytes do not exist in the record, the retrieval is still successful and any existing bytes are returned.

When storing a data item into the database, the **dlen** bytes starting **doff** bytes from the beginning of the specified key's data record are replaced by the data specified by the **data** and **size** fields. If **dlen** is smaller than **size**, the record will grow, and if **dlen** is larger than **size**, the record will shrink. If the specified bytes do not exist, the record will be extended using nul bytes as necessary, and the store call will still succeed.

The following are various examples of the put case for the DB_DBT_PARTIAL flag. In all examples, the initial data item is 20 bytes in length:

ABCDEFGHIJ0123456789

1.
```
size = 20
doff = 0
dlen = 20
data = abcdefghijabcdefghij
```

```
Result: The 20 bytes at offset 0 are replaced by the 20 bytes of
data; that is, the entire record is replaced.

ABCDEFGHIJ0123456789 -> abcdefghijabcdefghij
```

2. ```
   size = 10
   doff = 20
   dlen = 0
   data = abcdefghij

   Result: The 0 bytes at offset 20 are replaced by the 10 bytes of
   data; that is, the record is extended by 10 bytes.

   ABCDEFGHIJ0123456789 -> ABCDEFGHIJ0123456789abcdefghij
   ```

3. ```
   size = 10
   doff = 10
   dlen = 5
   data = abcdefghij

   Result: The 5 bytes at offset 10 are replaced by the 10 bytes of
   data.

   ABCDEFGHIJ0123456789 -> ABCDEFGHIJabcdefghij56789
   ```

4. ```
   size = 10
   doff = 10
   dlen = 0
   data = abcdefghij

   Result: The 0 bytes at offset 10 are replaced by the 10 bytes of
   data; that is, 10 bytes are inserted into the record.

   ABCDEFGHIJ0123456789 -> ABCDEFGHIJabcdefghij0123456789
   ```

5. ```
   size = 10
   doff = 2
   dlen = 15
   data = abcdefghij

   Result: The 15 bytes at offset 2 are replaced by the 10 bytes of
   data.

   ABCDEFGHIJ0123456789 -> ABabcdefghij789
   ```

6. ```
   size = 10
   doff = 0
   ```
   ```

```
dlen = 0
data = abcdefghij

Result: The 0 bytes at offset 0 are replaced by the 10 bytes of
data; that is, the 10 bytes are inserted at the beginning of the
record.

ABCDEFGHIJ0123456789 -> abcdefghijABCDEFGHIJ0123456789
```

7.
```
size = 0
doff = 0
dlen = 10
data = ""

Result: The 10 bytes at offset 0 are replaced by the 0 bytes of
data; that is, the first 10 bytes of the record are discarded.

ABCDEFGHIJ0123456789 -> 0123456789
```

8.
```
size = 10
doff = 25
dlen = 0
data = abcdefghij

Result: The 0 bytes at offset 25 are replaced by the 10 bytes of
data; that is, 10 bytes are inserted into the record past the end
of the current data (\0 represents a nul byte).

ABCDEFGHIJ0123456789 -> ABCDEFGHIJ0123456789\0\0\0\0\0abcdefghij
```

Storing C/C++ structures/objects

Berkeley DB can store any kind of data, that is, it is entirely 8-bit clean. How you use this depends, to some extent, on the application language you are using. In the C/C++ languages, there are a couple of different ways to store structures and objects.

First, you can do some form of run-length encoding and copy your structure into another piece of memory before storing it:

```
struct {
 char *data1;
 u_int32_t data2;
 ...
} info;
size_t len;
u_int8_t *p, data_buffer[1024];

p = &data_buffer[0];
```

```
len = strlen(info.data1);
memcpy(p, &len, sizeof(len));
p += sizeof(len);
memcpy(p, info.data1, len);
p += len;
memcpy(p, &info.data2, sizeof(info.data2));
p += sizeof(info.data2);
...
```

and so on, until all the fields of the structure have been loaded into the byte array. If you want more examples, see the Berkeley DB logging routines (for example, btree/btree_auto.c:__bam_split_log()). This technique is generally known as "marshalling". If you use this technique, you must then un-marshall the data when you read it back:

```
struct {
 char *data1;
 u_int32_t data2;
 ...
} info;
size_t len;
u_int8_t *p;

p = &data_buffer[0];
memcpy(&len, p, sizeof(len));
p += sizeof(len);
info.data1 = malloc(len);
memcpy(info.data1, p, len);
p += len;
memcpy(&info.data2, p, sizeof(info.data2));
p += sizeof(info.data2);
...
```

and so on.

The second way to solve this problem only works if you have just one variable length field in the structure. In that case, you can declare the structure as follows:

```
struct {
 int a, b, c;
 u_int8_t buf[1];
} info;
```

Then, let's say you have a string you want to store in this structure. When you allocate the structure, you allocate it as:

```
malloc(sizeof(struct info) + strlen(string));
```

Since the allocated memory is contiguous, you can the initialize the structure as:

```
info.a = 1;
info.b = 2;
```

```
info.c = 3;
memcpy(&info.buf[0], string, strlen(string) + 1);
```

and give it to Berkeley DB to store, with a length of:

```
sizeof(struct info) + strlen(string);
```

In this case, the structure can be copied out of the database and used without any additional work.

Retrieved key/data permanence for C/C++

When using the non-cursor Berkeley DB calls to retrieve key/data items under the C/C++ APIs (for example, DB->get()), the memory to which the pointer stored into the DBT refers is only valid until the next call to Berkeley DB using the DB handle. (This includes **any** use of the returned DB handle, including by another thread of control within the process. For this reason, when multiple threads are using the returned DB handle concurrently, one of the DB_DBT_MALLOC, DB_DBT_REALLOC or DB_DBT_USERMEM flags must be specified with any non-cursor DBT used for key or data retrieval.)

When using the cursor Berkeley DB calls to retrieve key/data items under the C/C++ APIs (for example, DBC->get()), the memory to which the pointer stored into the DBT refers is only valid until the next call to Berkeley DB using the DBC returned by DB->cursor().

Error support

Berkeley DB offers programmatic support for displaying error return values.

The db_strerror() function returns a pointer to the error message corresponding to any Berkeley DB error return, similar to the ANSI C strerror function, but is able to handle both system error returns and Berkeley DB specific return values.

For example:

```
int ret;
if ((ret = dbp->put(dbp, NULL, &key, &data, 0)) != 0) {
 fprintf(stderr, "put failed: %s\n", db_strerror(ret));
 return (1);
}
```

There are also two additional error methods, DB->err() and `DB->errx()`. These methods work like the ANSI C X3.159-1989 (ANSI C) printf function, taking a printf-style format string and argument list, and writing a message constructed from the format string and arguments.

The DB->err() method appends the standard error string to the constructed message; the `DB->errx()` method does not. These methods provide simpler ways of displaying Berkeley DB error messages. For example, if your application tracks session IDs in a variable called session_id, it can include that information in its error messages:

Error messages can additionally be configured to always include a prefix (for example, the program name) using the DB->set_errpfx() method.

```
#define DATABASE "access.db"

int ret;

(void)dbp->set_errpfx(dbp, program_name);

if ((ret = dbp->open(dbp,
    NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, 0664)) != 0) {
 dbp->err(dbp, ret, "%s", DATABASE);
 dbp->errx(dbp,
  "contact your system administrator: session ID was %d",
      session_id);
 return (1);
}
```

For example, if the program were called my_app and the open call returned an EACCESS system error, the error messages shown would appear as follows:

```
my_app: access.db: Permission denied.
my_app: contact your system administrator: session ID was 14
```

Cursor stability

In the absence of locking, no guarantees are made about the stability of cursors in different threads of control. However, the Btree, Queue and Recno access methods guarantee that cursor operations, interspersed with any other operation in the same thread of control will always return keys in order and will return each non-deleted key/data pair exactly once. Because the Hash access method uses a dynamic hashing algorithm, it cannot guarantee any form of stability in the presence of inserts and deletes unless transactional locking is performed.

If locking was specified when the Berkeley DB environment was opened, but transactions are not in effect, the access methods provide repeatable reads with respect to the cursor. That is, a DB_CURRENT call on the cursor is guaranteed to return the same record as was returned on the last call to the cursor.

In the presence of transactions, the Btree, Hash and Recno access methods provide degree 3 isolation (serializable transactions). The Queue access method provides degree 3 isolation with the exception that it permits phantom records to appear between calls. That is, deleted records are not locked, therefore another transaction may replace a deleted record between two calls to retrieve it. The record would not appear in the first call but would be seen by the second call. For readers not enclosed in transactions, all access method calls provide degree 2 isolation, that is, reads are not repeatable. A transaction may be declared to run with degree 2 isolation by specifying the DB_READ_COMMITTED flag. Finally, Berkeley DB provides degree 1 isolation when the DB_READ_UNCOMMITTED flag is specified; that is, reads may see data modified in transactions which have not yet committed.

For all access methods, a cursor scan of the database performed within the context of a transaction is guaranteed to return each key/data pair once and only once, except in the following case. If, while performing a cursor scan using the Hash access method, the transaction

performing the scan inserts a new pair into the database, it is possible that duplicate key/data pairs will be returned.

Database limits

The largest database file that Berkeley DB can handle depends on the page size selected by the application. Berkeley DB stores database file page numbers as unsigned 32-bit numbers and database file page sizes as unsigned 16-bit numbers. Using the maximum database page size of 65536, this results in a maximum database file size of 2^{48} (256 terabytes). The minimum database page size is 512 bytes, which results in a minimum maximum database size of 2^{41} (2 terabytes).

The largest database file Berkeley DB can support is potentially further limited if the host system does not have filesystem support for files larger than 2^{32}, including the ability to seek to absolute offsets within those files.

The largest key or data item that Berkeley DB can support is 2^{32}, or more likely limited by available memory. Specifically, while key and data byte strings may be of essentially unlimited length, any one of them must fit into available memory so that it can be returned to the application. As some of the Berkeley DB interfaces return both key and data items to the application, those interfaces will require that any key/data pair fit simultaneously into memory. Further, as the access methods may need to compare key and data items with other key and data items, it may be a requirement that any two key or two data items fit into available memory. Finally, when writing applications supporting transactions, it may be necessary to have an additional copy of any data item in memory for logging purposes.

The maximum Btree depth is 255.

Disk space requirements

It is possible to estimate the total database size based on the size of the data. The following calculations are an estimate of how many bytes you will need to hold a set of data and then how many pages it will take to actually store it on disk.

Space freed by deleting key/data pairs from a Btree or Hash database is never returned to the filesystem, although it is reused where possible. This means that the Btree and Hash databases are grow-only. If enough keys are deleted from a database that shrinking the underlying file is desirable, you should create a new database and copy the records from the old one into it.

These are rough estimates at best. For example, they do not take into account overflow records, filesystem metadata information, large sets of duplicate data items (where the key is only stored once), or real-life situations where the sizes of key and data items are wildly variable, and the page-fill factor changes over time.

Btree

The formulas for the Btree access method are as follows:

```
useful-bytes-per-page = (page-size - page-overhead) * page-fill-factor
```

```
bytes-of-data = n-records *
    (bytes-per-entry + page-overhead-for-two-entries)


n-pages-of-data = bytes-of-data / useful-bytes-per-page


total-bytes-on-disk = n-pages-of-data * page-size
```

The **useful-bytes-per-page** is a measure of the bytes on each page that will actually hold the application data. It is computed as the total number of bytes on the page that are available to hold application data, corrected by the percentage of the page that is likely to contain data. The reason for this correction is that the percentage of a page that contains application data can vary from close to 50% after a page split to almost 100% if the entries in the database were inserted in sorted order. Obviously, the **page-fill-factor** can drastically alter the amount of disk space required to hold any particular data set. The page-fill factor of any existing database can be displayed using the db_stat utility.

The page-overhead for Btree databases is 26 bytes. As an example, using an 8K page size, with an 85% page-fill factor, there are 6941 bytes of useful space on each page:

```
6941 = (8192 - 26) * .85
```

The total **bytes-of-data** is an easy calculation: It is the number of key or data items plus the overhead required to store each item on a page. The overhead to store a key or data item on a Btree page is 5 bytes. So, it would take 1560000000 bytes, or roughly 1.34GB of total data to store 60,000,000 key/data pairs, assuming each key or data item was 8 bytes long:

```
1560000000 = 60000000 * ((8 + 5) * 2)
```

The total pages of data, **n-pages-of-data**, is the **bytes-of-data** divided by the **useful-bytes-per-page**. In the example, there are 224751 pages of data.

```
224751 = 1560000000 / 6941
```

The total bytes of disk space for the database is **n-pages-of-data** multiplied by the **page-size**. In the example, the result is 1841160192 bytes, or roughly 1.71GB.

```
1841160192 = 224751 * 8192
```

Hash

The formulas for the Hash access method are as follows:

```
useful-bytes-per-page = (page-size - page-overhead)


bytes-of-data = n-records *
    (bytes-per-entry + page-overhead-for-two-entries)


n-pages-of-data = bytes-of-data / useful-bytes-per-page
```

```
total-bytes-on-disk = n-pages-of-data * page-size
```

The **useful-bytes-per-page** is a measure of the bytes on each page that will actually hold the application data. It is computed as the total number of bytes on the page that are available to hold application data. If the application has explicitly set a page-fill factor, pages will not necessarily be kept full. For databases with a preset fill factor, see the calculation below. The page-overhead for Hash databases is 26 bytes and the page-overhead-for-two-entries is 6 bytes.

As an example, using an 8K page size, there are 8166 bytes of useful space on each page:

```
8166 = (8192 - 26)
```

The total **bytes-of-data** is an easy calculation: it is the number of key/data pairs plus the overhead required to store each pair on a page. In this case that's 6 bytes per pair. So, assuming 60,000,000 key/data pairs, each of which is 8 bytes long, there are 1320000000 bytes, or roughly 1.23GB of total data:

```
1320000000 = 60000000 * (16 + 6)
```

The total pages of data, **n-pages-of-data**, is the **bytes-of-data** divided by the **useful-bytes-per-page**. In this example, there are 161646 pages of data.

```
161646 = 1320000000 / 8166
```

The total bytes of disk space for the database is **n-pages-of-data** multiplied by the **page-size**. In the example, the result is 1324204032 bytes, or roughly 1.23GB.

```
1324204032 = 161646 * 8192
```

Now, let's assume that the application specified a fill factor explicitly. The fill factor indicates the target number of items to place on a single page (a fill factor might reduce the utilization of each page, but it can be useful in avoiding splits and preventing buckets from becoming too large). Using our estimates above, each item is 22 bytes (16 + 6), and there are 8166 useful bytes on a page (8192 - 26). That means that, on average, you can fit 371 pairs per page.

```
371 = 8166 / 22
```

However, let's assume that the application designer knows that although most items are 8 bytes, they can sometimes be as large as 10, and it's very important to avoid overflowing buckets and splitting. Then, the application might specify a fill factor of 314.

```
314 = 8166 / 26
```

With a fill factor of 314, then the formula for computing database size is

```
n-pages-of-data = npairs / pairs-per-page
```

or 191082.

```
191082 = 60000000 / 314
```

At 191082 pages, the total database size would be 1565343744, or 1.46GB.

```
1565343744 = 191082 * 8192
```

There are a few additional caveats with respect to Hash databases. This discussion assumes that the hash function does a good job of evenly distributing keys among hash buckets. If the function does not do this, you may find your table growing significantly larger than you expected. Secondly, in order to provide support for Hash databases coexisting with other databases in a single file, pages within a Hash database are allocated in power-of-two chunks. That means that a Hash database with 65 buckets will take up as much space as a Hash database with 128 buckets; each time the Hash database grows beyond its current power-of-two number of buckets, it allocates space for the next power-of-two buckets. This space may be sparsely allocated in the file system, but the files will appear to be their full size. Finally, because of this need for contiguous allocation, overflow pages and duplicate pages can be allocated only at specific points in the file, and this too can lead to sparse hash tables.

Specifying a Berkeley DB schema using SQL DDL

When starting a new Berkeley DB project, much of the code that you must write is dedicated to defining the BDB environment: what databases it contains, the types of the databases, and so forth. Also, since records in BDB are just byte arrays, you must write code that assembles and interprets these byte arrays.

Much of this code can be written automatically (in C) by the db_sql utility. To use it, you first specify the schema of your Berkeley DB environment in SQL Data Definition Language (DDL). Then you invoke the db_sql command, giving the DDL as input. **db_sql** reads the DDL, and writes C code that implements a storage-layer API suggested by the DDL.

The generated API includes a general-purpose initialization function, which sets up the environment and the databases (creating them if they don't already exist). It also includes C structure declarations for each record type, and numerous specialized functions for storing and retrieving those records.

db_sql can also produce a simple test program that exercises the generated API. This program is useful as an example of how to use the API. It contains calls to all of the interface functions, along with commentary explaining what the code is doing.

Once the storage layer API is produced, your application may use it as is, or you may customize it as much as you like by editing the generated source code. Be warned, however: **db_sql** is a one-way process; there is no way to automatically incorporate customizations into newly generated code, if you decide to run **db_sql** again.

To learn more about **db_sql**, please consult the db_sql utility manual page in the Berkeley DB C API guide.

Access method tuning

There are a few different issues to consider when tuning the performance of Berkeley DB access method applications.

access method

An application's choice of a database access method can significantly affect performance. Applications using fixed-length records and integer keys are likely to get better performance from the Queue access method. Applications using variable-length

records are likely to get better performance from the Btree access method, as it tends to be faster for most applications than either the Hash or Recno access methods. Because the access method APIs are largely identical between the Berkeley DB access methods, it is easy for applications to benchmark the different access methods against each other. See Selecting an access method (page 15) for more information.

cache size

The Berkeley DB database cache defaults to a fairly small size, and most applications concerned with performance will want to set it explicitly. Using a too-small cache will result in horrible performance. The first step in tuning the cache size is to use the db_stat utility (or the statistics returned by the DB->stat() function) to measure the effectiveness of the cache. The goal is to maximize the cache's hit rate. Typically, increasing the size of the cache until the hit rate reaches 100% or levels off will yield the best performance. However, if your working set is sufficiently large, you will be limited by the system's available physical memory. Depending on the virtual memory and file system buffering policies of your system, and the requirements of other applications, the maximum cache size will be some amount smaller than the size of physical memory. If you find that the db_stat utility shows that increasing the cache size improves your hit rate, but performance is not improving (or is getting worse), then it's likely you've hit other system limitations. At this point, you should review the system's swapping/paging activity and limit the size of the cache to the maximum size possible without triggering paging activity. Finally, always remember to make your measurements under conditions as close as possible to the conditions your deployed application will run under, and to test your final choices under worst-case conditions.

shared memory

By default, Berkeley DB creates its database environment shared regions in filesystem backed memory. Some systems do not distinguish between regular filesystem pages and memory-mapped pages backed by the filesystem, when selecting dirty pages to be flushed back to disk. For this reason, dirtying pages in the Berkeley DB cache may cause intense filesystem activity, typically when the filesystem sync thread or process is run. In some cases, this can dramatically affect application throughput. The workaround to this problem is to create the shared regions in system shared memory (DB_SYSTEM_MEM) or application private memory (DB_PRIVATE), or, in cases where this behavior is configurable, to turn off the operating system's flushing of memory-mapped pages.

large key/data items

Storing large key/data items in a database can alter the performance characteristics of Btree, Hash and Recno databases. The first parameter to consider is the database page size. When a key/data item is too large to be placed on a database page, it is stored on "overflow" pages that are maintained outside of the normal database structure (typically, items that are larger than one-quarter of the page size are deemed to be too large). Accessing these overflow pages requires at least one additional page reference over a normal access, so it is usually better to increase the page size than to create a database with a large number of overflow pages. Use the db_stat utility (or the statistics returned by the DB->stat() method) to review the number of overflow pages in the database.

The second issue is using large key/data items instead of duplicate data items. While this can offer performance gains to some applications (because it is possible to retrieve several data items in a single get call), once the key/data items are large enough to be pushed off-page, they will slow the application down. Using duplicate data items is usually the better choice in the long run.

A common question when tuning Berkeley DB applications is scalability. For example, people will ask why, when adding additional threads or processes to an application, the overall database throughput decreases, even when all of the operations are read-only queries.

First, while read-only operations are logically concurrent, they still have to acquire mutexes on internal Berkeley DB data structures. For example, when searching a linked list and looking for a database page, the linked list has to be locked against other threads of control attempting to add or remove pages from the linked list. The more threads of control you add, the more contention there will be for those shared data structure resources.

Second, once contention starts happening, applications will also start to see threads of control convoy behind locks (especially on architectures supporting only test-and-set spin mutexes, rather than blocking mutexes). On test-and-set architectures, threads of control waiting for locks must attempt to acquire the mutex, sleep, check the mutex again, and so on. Each failed check of the mutex and subsequent sleep wastes CPU and decreases the overall throughput of the system.

Third, every time a thread acquires a shared mutex, it has to shoot down other references to that memory in every other CPU on the system. Many modern snoopy cache architectures have slow shoot down characteristics.

Fourth, schedulers don't care what application-specific mutexes a thread of control might hold when de-scheduling a thread. If a thread of control is descheduled while holding a shared data structure mutex, other threads of control will be blocked until the scheduler decides to run the blocking thread of control again. The more threads of control that are running, the smaller their quanta of CPU time, and the more likely they will be descheduled while holding a Berkeley DB mutex.

The results of adding new threads of control to an application, on the application's throughput, is application and hardware specific and almost entirely dependent on the application's data access pattern and hardware. In general, using operating systems that support blocking mutexes will often make a tremendous difference, and limiting threads of control to to some small multiple of the number of CPUs is usually the right choice to make.

Access method FAQ

1. **Is a Berkeley DB database the same as a "table"?**

   Yes; "tables" are databases, "rows" are key/data pairs, and "columns" are application-encapsulated fields within a data item (to which Berkeley DB does not directly provide access).

2. **I'm getting an error return in my application, but I can't figure out what the library is complaining about.**

See DB_ENV->set_errcall(), DB_ENV->set_errfile() and DB->set_errfile() for ways to get additional information about error returns from Berkeley DB.

3. **Are Berkeley DB databases portable between architectures with different integer sizes and different byte orders ?**

   Yes. Specifically, databases can be moved between 32- and 64-bit machines, as well as between little- and big-endian machines. See Selecting a byte order (page 21) for more information.

4. **I'm seeing database corruption when creating multiple databases in a single physical file.**

   This problem is usually the result of DB handles not sharing an underlying database environment. See Opening multiple databases in a single file (page 40) for more information.

5. **I'm using integers as keys for a Btree database, and even though the key/data pairs are entered in sorted order, the page-fill factor is low.**

   This is usually the result of using integer keys on little-endian architectures such as the x86. Berkeley DB sorts keys as byte strings, and little-endian integers don't sort well when viewed as byte strings. For example, take the numbers 254 through 257. Their byte patterns on a little-endian system are:

   ```
   254 fe 0 0 0
   255 ff 0 0 0
   256  0 1 0 0
   257  1 1 0 0
   ```

   If you treat them as strings, then they sort badly:

   ```
   256
   257
   254
   255
   ```

   On a big-endian system, their byte patterns are:

   ```
   254 0 0 0 fe
   255 0 0 0 ff
   256 0 0 1 0
   257 0 0 1 1
   ```

   and so, if you treat them as strings they sort nicely. Which means, if you use steadily increasing integers as keys on a big-endian system Berkeley DB behaves well and you get compact trees, but on a little-endian system Berkeley DB produces much less compact trees. To avoid this problem, you may want to convert the keys to flat text or big-endian representations, or provide your own Btree comparison (page 23)

6. **Is there any way to avoid double buffering in the Berkeley DB system?**

   While you cannot avoid double buffering entirely, there are a few things you can do to address this issue:

First, the Berkeley DB cache size can be explicitly set. Rather than allocate additional space in the Berkeley DB cache to cover unexpectedly heavy load or large table sizes, double buffering may suggest you size the cache to function well under normal conditions, and then depend on the file buffer cache to cover abnormal conditions. Obviously, this is a trade-off, as Berkeley DB may not then perform as well as usual under abnormal conditions.

Second, depending on the underlying operating system you're using, you may be able to alter the amount of physical memory devoted to the system's file buffer cache. Altering this type of resource configuration may require appropriate privileges, or even operating system reboots and/or rebuilds, on some systems.

Third, changing the size of the Berkeley DB environment regions can change the amount of space the operating system makes available for the file buffer cache, and it's often worth considering exactly how the operating system is dividing up its available memory. Further, moving the Berkeley DB database environment regions from filesystem backed memory into system memory (or heap memory), can often make additional system memory available for the file buffer cache, especially on systems without a unified buffer cache and VM system.

Finally, for operating systems that allow buffering to be turned off, specifying the DB_DIRECT_DB and DB_LOG_DIRECT flags will attempt to do so.

7. **I'm seeing database corruption when I run out of disk space.**

   Berkeley DB can continue to run when when out-of-disk-space errors occur, but it requires the application to be transaction protected. Applications which do not enclose update operations in transactions cannot recover from out-of-disk-space errors, and the result of running out of disk space may be database corruption.

8. **How can I associate application information with a DB or DB_ENV handle?**

   In the C API, the DB and DB_ENV structures each contain an "app_private" field intended to be used to reference application-specific information. See the db_create() and db_env_create() documentation for more information.

   In the C++ or Java APIs, the easiest way to associate application-specific data with a handle is to subclass the Db or DbEnv, for example subclassing Db to get MyDb. Objects of type MyDb will still have the Berkeley DB API methods available on them, and you can put any extra data or methods you want into the MyDb class. If you are using "callback" APIs that take Db or DbEnv arguments (for example, DB->set_bt_compare()) these will always be called with the Db or DbEnv objects you create. So if you always use MyDb objects, you will be able to take the first argument to the callback function and cast it to a MyDb (in C++, cast it to (MyDb*)). That will allow you to access your data members or methods.

Chapter 5. Java API

Java configuration

Building the Berkeley DB java classes, the examples and the native support library is integrated into the normal build process. See Configuring Berkeley DB (page 290) and Building the Java API (page 314) for more information.

We expect that you already installed the Java JDK or equivalent on your system. For the sake of discussion, we assume that it is in a directory called db-VERSION; for example, you downloaded a Berkeley DB archive, and you did not change the top-level directory name. The files related to Java are in three subdirectories of db-VERSION: java (the java source files), libdb_java (the C++ files that provide the "glue" between java and Berkeley DB) and examples_java (containing all examples code). The directory tree looks like this:

```
db-VERSION
|-- java
|   `-- src
|       `-- com
|           `-- sleepycat
|               |-- bind
|               |-- db
|               |   `-- ...
|               `-- util
|-- examples_java
|   `-- src
|       `-- db
|           `-- ...
`-- libdb_java
    `-- ...
```

This naming conforms to the de facto standard for naming java packages. When the java code is built, it is placed into two jar files: `db.jar`, containing the db package, and `dbexamples.jar`, containing the examples.

For your application to use Berkeley DB successfully, you must set your `CLASSPATH` environment variable to include the full pathname of the db jar files as well as the classes in your java distribution. On UNIX, `CLASSPATH` is a colon-separated list of directories and jar files; on Windows, it is separated by semicolons. On UNIX, the jar files are put in your build directory, and when you do the make install step, they are copied to the lib directory of your installation tree. On Windows, the jar files are placed in the Release or Debug subdirectory with your other objects.

The Berkeley DB Java classes are mostly implemented in native methods. Before you can use them, you need to make sure that the DLL or shared library containing the native methods can be found by your Java runtime. On Windows, you should set your PATH variable to include:

```
db-VERSION\build_windows\Release
```

On UNIX, you should set the LD_LIBRARY_PATH environment variable or local equivalent to include the Berkeley DB library installation directory. Of course, the standard install directory may have been changed for your site; see your system administrator for details.

On other platforms, the path can be set on the command line as follows (assuming the shared library is in /usr/local/BerkeleyDB/lib:)

```
% java -Djava.library.path=/usr/local/BerkeleyDB/lib ...
```

Regardless, if you get the following exception when you run, you probably do not have the library search path configured correctly:

```
java.lang.UnsatisfiedLinkError
```

Different Java interpreters provide different error messages if the CLASSPATH value is incorrect, a typical error is the following:

```
java.lang.NoClassDefFoundError
```

To ensure that everything is running correctly, you may want to try a simple test from the example programs in

```
db-VERSION/examples_java/src/db
```

For example, the following sample program will prompt for text input lines, which are then stored in a Btree database named access.db in your current directory:

```
% java db.AccessExample
```

Try giving it a few lines of input text and then end-of-file. Before it exits, you should see a list of the lines you entered display with data items. This is a simple check to make sure the fundamental configuration is working correctly.

Compatibility

The Berkeley DB Java API has been tested with the Sun Microsystem's JDK 1.5 (Java 5) on Linux, Windows and OS X. It should work with any JDK 1.5- compatible environment.

Java programming notes

Although the Java API parallels the Berkeley DB C++/C interface in many ways, it differs where the Java language requires. For example, the handle method names are camel-cased and conform to Java naming patterns. (The C++/C method names are currently provided, but are deprecated.)

1. The Java runtime does not automatically close Berkeley DB objects on finalization. There are several reasons for this. One is that finalization is generally run only when garbage collection occurs, and there is no guarantee that this occurs at all, even on exit. Allowing specific Berkeley DB actions to occur in ways that cannot be replicated seems wrong. Second, finalization of objects may happen in an arbitrary order, so we would have to do extra bookkeeping to make sure that everything was closed in the proper order. The best word of advice is to always do a close() for any matching open() call. Specifically, the Berkeley DB

package requires that you explicitly call close on each individual Database [../java/com/sleepycat/db/Database.html] and Cursor [../java/com/sleepycat/db/Cursor.html] object that you opened. Your database activity may not be synchronized to disk unless you do so.

2. Some methods in the Java API have no return type, and throw a DatabaseException [../java/com/sleepycat/db/DatabaseException.html] when an severe error arises. There are some notable methods that do have a return value, and can also throw an exception. The "get" methods in Database [../java/com/sleepycat/db/Database.html] and Cursor [../java/com/sleepycat/db/Cursor.html] both return 0 when a get succeeds, DB_NOTFOUND (page 230) when the key is not found, and throw an error when there is a severe error. This approach allows the programmer to check for typical data-driven errors by watching return values without special casing exceptions.

   An object of type MemoryException [../java/com/sleepycat/db/MemoryException.html] is thrown when a Dbt is too small to hold the corresponding key or data item.

   An object of type DeadlockException [../java/com/sleepycat/db/DeadlockException.html] is thrown when a deadlock would occur.

   An object of type RunRecoveryException [../java/com/sleepycat/db/RunRecoveryException.html], a subclass of DatabaseException [../java/com/sleepycat/db/DatabaseException.html], is thrown when there is an error that requires a recovery of the database using db_recover utility.

   An object of type IllegalArgumentException [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/IllegalArgumentException.html] a standard Java Language exception, is thrown when there is an error in method arguments.

   An object of type OutOfMemoryError [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/OutOfMemoryError.html] is thrown when the system cannot provide enough memory to complete the operation (the ENOMEM system error on UNIX).

3. If there are embedded nulls in the **curslist** argument for Database.join(com.sleepycat.db.Cursor[], com.sleepycat.db.JoinConfig) [../java/com/sleepycat/db/Database.html#join(com.sleepycat.db.Cursor[], com.sleepycat.db.JoinConfig)], they will be treated as the end of the list of cursors, even if you may have allocated a longer array. Fill in all the cursors in your array unless you intend to cut it short.

4. If you are using custom class loaders in your application, make sure that the Berkeley DB classes are loaded by the system class loader, not a custom class loader. This is due to a JVM bug that can cause an access violation during finalization (see the bug 4238486 in Sun Microsystem's Java Bug Database).

Java FAQ

1. **On what platforms is the Berkeley DB Java API supported?**

   All platforms supported by Berkeley DB that have a JVM compatible with J2SE 1.4 or above.

2. **How does the Berkeley DB Java API relate to the J2EE standard?**

The Berkeley DB Java API does not currently implement any part of the J2EE standard. That said, it does implement the implicit standard for Java Java Collections [http://java.sun.com/j2se/1.5.0/docs/guide/collections/]. The concept of a transaction exists in several Java packages (J2EE, XA, JINI to name a few). Support for these APIs will be added based on demand in future versions of Berkeley DB.

3. **How should I incorporate db.jar and the db native library into a Tomcat or other J2EE application servers?**

   Tomcat and other J2EE application servers have the ability to rebuild and reload code automatically. When using Tomcat this is the case when "reloadable" is set to "true". If your WAR file includes the db.jar it too will be reloaded each time your code is reloaded. This causes exceptions as the native library can't be loaded more than once and there is no way to unload native code. The solution is to place the db.jar in $TOMCAT_HOME/common/lib and let Tomcat load that library once at start time rather than putting it into the WAR that gets reloaded over and over.

4. **Can I use the Berkeley DB Java API from within a EJB, a Servlet or a JSP page?**

   Yes. The Berkeley DB Java API can be used from within all the popular J2EE application servers in many different ways.

5. **During one of the first calls to the Berkeley DB Java API, a DbException is thrown with a "Bad file number" or "Bad file descriptor" message.**

   There are known large-file support bugs under JNI in various releases of the JDK. Please upgrade to the latest release of the JDK, and, if that does not solve the problem, disable big file support using the --disable-largefile configuration option.

6. **How can I use native methods from a debug build of the Java library?**

   Set Java's library path so that the debug version of Berkeley DB's Java library appears, but the release version does not. Berkeley DB tries to load the release library first, and if that fails tries the debug library.

7. **Why is ClassNotFoundException thrown when adding a record to the database, when a SerialBinding is used?**

   This problem occurs if you copy the db.jar file into the Java extensions (ext) directory. This will cause the database code to run under the System class loader, and it won't be able to find your application classes.

   You'll have to actually remove db.jar from the Java extension directory. If you have more than one installation of Java, be sure to remove it from all of them. This is necessary even if db.jar is specified in the classpath.

   An example of the exception is:

```
collections.ship.basic.SupplierKey
at java.net.URLClassLoader$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
```

```
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClassInternal(Unknown Source)
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Unknown Source)
at com.sleepycat.bind.serial.StoredClassCatalog.getClassInfo(StoredClassCatalog.java:211)
...
```

8. **I'm upgrading my Java application to Berkeley DB 4.3. Can I use the com.sleepycat.db.internal package rather than porting my code to the new API?**

   While it is possible to use the low-level API from applications, there are some caveats that should be considered when upgrading. The first is that the internal API depends on some classes in the public API such as DatabaseEntry.

   In addition, the internal API is closer to the C API and doesn't have some of the default settings that were part of the earlier Java API. For example, applications will need to set the DB_THREAD flag explicitly if handles are to be used from multiple threads, or subtle errors may occur.

Chapter 6. C# API

A separate Visual Studio solution is provided to build the Berkeley DB C# classes, the examples and the native support library.

The C# API requires version 2.0 of the .NET framework and expects that it has already been installed on your system. For the sake of discussion, we assume that the Berkeley DB source is in a directory called db-*VERSION*; for example, you downloaded a Berkeley DB archive, and you did not change the top-level directory name. The files related to C# are in four subdirectories of db-*VERSION*: csharp (the C# source files), libdb_csharp (the C++ files that provide the "glue" between C# and Berkeley DB,) examples_csharp (containing all example code) and test\scr037 (containing NUnit tests for the API).

Building the C# API produces a managed assembly `libdb_dotnet`*`VERSION`*`.dll`, containing the API, and two native libraries: `libdb_csharp`*`VERSION`*`.dll` and `libdb`*`VERSION`*`.dll`. (For all three files, *VERSION* is [MAJOR][MINOR], i.e. for version 4.8 the managed assembly is `libdb_dotnet48.dll`.) Following the existing convention, native libraries are placed in either `db-`*`VERSION`*`\build_windows\Win32`or `db-`*`VERSION`*`\build_windows\x64`, depending upon the platform being targeted. In all cases, the managed assembly will be placed in `db-`*`VERSION`*`\build_windows\AnyCPU`.

Because the C# API uses P/Invoke, for your application to use Berkeley DB successfully, the .NET framework needs to be able to locate the native libaries. This means the native libraries need to either be copied to your application's directory, the Windows or System directory, or the location of the libraries needs to be added to the `PATH` environment variable. See the MSDN documentation of the DllImport attribute and Dynamic-Link Library Search Order for further information.

If you get the following exception when you run, the .NET platform probably is unable to locate the native libraries:

```
System.TypeInitializationException
```

To ensure that everything is running correctly, you may want to try a simple test from the example programs in the `db-`*`VERSION`*`\examples_csharp` directory.

For example, the ex_access sample program will prompt for text input lines, which are then stored in a Btree database named `access.db`. It is designed to be run from either the `db-`*`VERSION`*`\build_windows\Debug` or `db-`*`VERSION`*`\build_windows\Release` directory. Try giving it a few lines of input text and then a blank line. Before it exits, you should see a list of the lines you entered display with data items. This is a simple check to make sure the fundamental configuration is working correctly.

Compatibility

The Berkeley DB C# API has been tested with the Microsoft .NET Platform (version 2.0) on Windows.

Chapter 7. Standard Template Library API

Dbstl introduction

Dbstl is a C++ STL style API that provides for Berkeley DB usage. It allows for the storage and retrieval of data/objects of any type using Berkeley DB databases, but with an interface that mimics that of C++ STL containers. Dbstl provides access to all of the functionality of Berkeley DB available via this STL-style API.

With proper configuration, dbstl is able to store/retrieve any complex data types. There is no need to perform repetitive marshalling and unmarshalling of data. Dbstl also properly manages the life-cycle of all Berkeley DB structures and objects.

Standards compatible

Dbstl is composed of many container and iterator class templates. These containers and iterators correspond exactly to each container and iterator available in the C++ STL API, including identical sets of methods. This allows existing algorithms, functions and container-adapters for C++ STL to use dbstl containers through its standard iterators. This means that existing STL code can manipulate Berkeley DB databases. As a result, existing C++ STL code can very easily use dbstl to gain persistence and transaction guarantees.

Performance overhead

Because dbstl uses C++ template technologies, its performance overhead is minimal.

The dbstl API performs almost equally to the C API, as measured by two different implementations of the TPC-B benchmark: `ex_tpcb` and `exstl_tpcb`.

Portability

The degree to which dbstl is portable to a new platform is determined by whether Berkeley DB is available on the platform, as well as whether an appropriate C++ compiler is available on the platform.

For information on porting Berkeley DB to new platforms, see the *Berkeley DB Porting Guide*.

Almost all the advanced C++ template features are used in dbstl, including:

- member function templates
- member function template overloads
- partial specialization
- default template parameters.

For this reason, you need a standards-compatible C++ compiler to build dbstl. As of this writing, the following compilers are known to build dbstl successfully:

- MSVC8
- gcc3.4.4 and above
- Intel C++ 9 and above

For *nix platforms, if you can successfully configure your Berkeley DB build script with `--enable-stl`, then you should be able to successfully build dbstl library and application code using it.

Besides its own test suite, dbstl has also been tested against, and passes, the following test suites:

- MS STL test suite
- SGI STL test suite

Dbstl typical use cases

Among others, the following are some typical use cases where dbstl would be prefered over C++ STL:

- Working with a large amount of data, more than can reside in memory. Using C++ STL would force a number of page swaps, which will degrade performance. When using dbstl, data is stored in a database and Berkeley DB ensures the needed data is in memory, so that the overall performance of the machine is not slowed down.
- Familiar Interface. dbstl provides a familiar interface to Berkeley DB, hiding the marshalling and unmashalling details and automatically managing Berkeley DB structures and objects.
- Transaction semantics. dbstl provides the ACID properties (or a subset of the ACID properties) in addition to supporting all of the STL functionality.
- Concurrent access. Few (if any) existing C++ STL implementations support reading/writing to the same container concurrently, dbstl does.
- Object persistence. dbstl allows your application to store objects in a database, and use the objects across different runs of your application. dbstl is capable of storing complicated objects which are not located in a contiguous chunk of memory, with some user configurations.

Dbstl examples

Because dbstl is so much like C++ STL, its usage exactly mirrors that of C++ STL, with the exception of a few optional Berkeley DB specific configurations. In fact, the only difference between a program using dbstl and one using C++ STL is the class names. That is, `vector` becomes `db_vector`, and `map` becomes `db_map`.

The typical procedure for using dbstl is:

1. Optionally create and open your own Berkeley DB environment and database handles using the DB C++ API. If you perform these opens using the C++ API, make sure to perform necessary environment and database configurations at that time.

2. Optionally pass environment and database handles to dbstl container constructors when you create dbstl container objects. Note that you can create a dbstl container without passing it an environment and database object. When you do this, an internal anonymous database is created for you. In this situation, dbstl provides no data persistence guarantees.

3. Perform dbstl-specific configurations. For example, you can configure cursor open flags, as well as database access for autocommit. You can also configure callback functions.

4. Interact with the data contained in your Berkeley DB databases using dbstl containers and iterators. This usage of dbstl is identical to C++ STL container and iterator usage.

5. At this time, you can also use dbstl calls that are specific to Berkeley DB. For example, you can use Berkeley DB specific calls that manage transaction begin/commit/abort, handle registration, and so forth. While these calls are part of dbstl, they have no equivalence in the C++ STL APIs.

6. When your application is done using Berkeley DB, you do not need to explicitly close any Berkeley DB handles (environments, database, cursors, and so forth). Dbstl automatically closes all such handles for you.

For examples of dbstl usage, see the example programs in the `$db/examples_stl` directory.

The following program listing provides two code fragments. You can find more example code in the `dbstl/examples/` and `dbstl/test` directories.

```
//////////////// Code Snippet 1 ////////////////
db_vector<int, ElementHolder<int> > vctr(100);
for (int i = 0; i < 100; i++)
 vctr[i] = i;

for (int i = 0; i < 100; i++) {
 cout<<"\nvctr["<<i<<"] : "<<vctr[i];
 vctr[i] = vctr[i] * vctr[i];
 cout<<"\nvctr["<<i<<"] squre : "<<vctr[i];
}

//////////////// Code Snippet 2 ////////////////
typedef db_map<char *, const char *, ElementHolder<const char *> >
        strmap_t2;
strmap_t2 strmap;
char str[2], str2[2];
str[1] = str2[1] = '\0';
for (char c = 0; c < 26; c++) {
 str[0] = c + 'a';
 str2[0] = 'z' - c;
 strmap[str] = str2;
```

```
}
for (strmap_t2::iterator itr = strmap.begin(); itr != strmap.end(); ++itr)
 cout<<endl<<itr->first<<" : "<<itr->second;

using namespace dbstl;
dbstl::db_map<char, int> v;
v['i'] = 1;
cout<<v['i'];


dbstl::db_map<char *, char *> name_addr_map;
// The strings rather than the memory pointers are stored into DB.
name_addr_map["Alex"] = "Sydney Australia";
name_addr_map["David"] = "Shenzhen China";
cout<<"Alex's address:"<<name_addr_map["Alex"];

dbstl::db_vector<Person> vi;
// Some callback configurations follow here.

// The strings and objects rather than pointers are stored into DB.

Person obj("David Zhao", "Oracle", new Office("Boston", "USA"));
vi.push_back(obj); // More person storage.
for (int I = 0; I < vi.size(); I++)
 cout<<vi[I];
```

The first snippet initializes a db_vector container of 100 elements, with an in-memory anonymous database internally created by dbstl. The only difference between this and C++ STL is dbstl requires one more `type` parameter: *`ElementHolder<int>`*. The `ElementHolder` class template should be used for every type of dbstl container that will store C++ primitive data types, such as `int`, `float`, `char *`, `wchar_t *`, and so forth. But these class templates should not be used for class types for reasons that we explain in the following chapters.

In the second code snippet, the assignment:

```
strmap[str] = str2;
```

is used to store a string pair (`(str, str2)`) instead of pointers to the underlying database.

The rest of the code used in these snippets is identical to the code you would use for C++ STL containers. However, by using dbstl, you are storing data into a Berkeley DB database. If you create your own database with backing files on disk, your data or objects can persist and be restored when the program runs again.

Berkeley DB configuration

While dbstl behaves like the C++ STL APIs in most situations, there are some Berkeley DB configuration activities that you can and should perform using dbstl. These activities are described in the following sections.

Registering database and environment handles

Remember the following things as you use Berkeley DB Database and Environment handles with dbstl:

- If you share environment or database handles among multiple threads, remember to specify the DB_THREAD flag in the open call to the handle.
- If you create or open environment and/or database handles without using the dbstl helper functions, `dbstl::open_db()` or `dbstl::open_env()`, remember that your environment and database handles should be:

  1. Allocated in the heap via "new" operator.
  2. Created using the DB_CXX_NO_EXCEPTIONS flag.
  3. In each thread sharing the handles, the handles are registered using either `dbstl::register_db()` or `dbstl::register_dbenv()`.

- If you opened the database or environment handle using the `open_db()` or `open_env()` functions, the thread opening the handles should not call `register_db()` or `register_env()` again. This is because they have already been registered by the `open_db()` or `open_env()` functions. However, other threads sharing these handles still must register them locally.

Truncate requirements

Some Berkeley DB operations require there to be no open cursors on the database handle at the time the operation occurs. Dbstl is aware of these requirements, and will attempt to close the cursors opened in the current thread when it performs these operations. However, the scope of dbstl's activities in this regard are limited to the current thread; it makes no attempt to close cursors opened in other threads. So you are required to ensure there are no open cursors on database handles shared across threads when operations are performed that require all cursors on that handle to be closed.

There are only a a few operations which require all open cursors to be closed. This include all container `clear()` and `swap()` functions, and all versions of `db_vection<>::assign()` functions. These functions require all cursors to be closed for the database because by default they remove all key/data pairs from the database by truncating it.

When a function removes all key/data pairs from a database, there are two ways it can perform this activity:

- The default method is to truncate the database, which is an operation that requires all cursors to be closed. As mentioned above, it is your responsibility to close cursors opened in other threads before performing this operation. Otherwise, the operation will fail.
- Alternatively, you can specify that the database not be truncated. Instead, you can cause dbstl to delete all key/data pairs individually, one after another. In this situation, open cursors in the database will not cause the delete operations to fail. However, due to lock

contention, the delete operations might not complete until all cursors are closed, which is when all their read locks are released.

Auto commit support

Dbstl supports auto commit for some of its container's operations. When a dbstl container is created using a `Db` or `DbEnv` object, if that object was opened using the DB_AUTO_COMMIT flag, then every operation subsequently performed on that object will be automatically enclosed in a unique transaction (unless the operation is already in an external transaction). This is identical to how the Berkeley DB C, C++ and Java APIs behave.

Note that only a subset of a container's operations support auto commit. This is because those operations that accept or return an iterator have to exist in an external transactional context and so cannot support auto commit.

The dbstl API documentation identifies when a method supports auto commit transactions.

Database and environment identity checks

When a container member function involves another container (for example, `db_vector::swap(self& v2)`), the two containers involved in the operation must not use the same database. Further, if the function is in an external or internal transaction context, then both containers must belong to the same transactional database environment; Otherwise, the two containers can belong to the same database environment, or two different ones.

For example, if `db_vector::swap(self& v2)` is an auto commit method or it is in an external transaction context, then `v2` must be in the same transactional database environment as this container, because a transaction is started internally that must be used by both `v2` and this container. If this container and the `v2` container have different database environments, and either of them are using transactions, an exception is thrown. This condition is checked in every such member function.

However, if the function is not in a transactional context, then the databases used by these containers can be in different environments because in this situation dbstl makes no attempt to wrap container operations in a common transaction context.

Products, constructors and configurations

You can use dbstl with all Berkeley DB products (DS, CDS, TDS, and HA). Because dbstl is a Berkeley DB interface, all necessary configurations for these products are performed using Berkeley DB's standard create/open/set APIs.

As a result, the dbstl container constructors differ from those of C++ STL because in dbstl no configuration is supported using the container constructors. On the other hand, dbstl container constructors accept already opened and configured environment and database handles. They also provide functions to retrieve some handle configuration, such as key comparison and hash functions, as required by the C++ STL specifications.

The constructors verify that the handles passed to them are well configured. This means they ensure that no banned settings are used, as well as ensuring that all required setting are performed. If the handles are not well configured, an `InvalidArgumentException` is thrown.

If a container constructor is not passed a database or environment handle, an internal anonymous database is created for you by dbstl. This anonymous database does not provide data persistence.

Using advanced Berkeley DB features with dbstl

This section describes advanced Berkeley DB features that are available through dbstl.

Using bulk retrieval iterators

Bulk retrieval is an optimization option for const iterators and nonconst but read-only iterators. Bulk retrieval can minimize the number of database accesses performed by your application. It does this by reading multiple entries at a time, which reduces read overhead. Note that non-sequential reads will benefit less from, or even be hurt by, this behavior, because it might result in unneeded data being read from the database. Also, non-serializable reads may read obsolete data, because part of the data read from the bulk read buffer may have been updated since the retrieval.

When using the default transaction isolation, iterators will perform serializable reads. In this situation, the bulk-retrieved data cannot be updated until the iterator's cursor is closed.

Iterators using a different isolation levels, such as DB_READ_COMMITTED or DB_READ_UNCOMMITTED will not perform serializable reads. The same is true for any iterators that do not use transactions.

A bulk retrieval iterator can only move in a singled direction, from beginning to end. This means that iterators only support operator++, and reverse iterators only support operator--.

Iterator objects that use bulk retrieval might contain hundreds of kilobytes of data, which makes copying the iterator object an expensive operation. If possible, use ++iterator rather than iterator++. This can save a useless copy construction of the iterator, as well as an unnecessary dup/close of the cursor.

You can configure bulk retrieval for each container using both in the const and non-const version of the `begin()` method. The non-const version of `begin()` will return a read-only cursor. Note that read-only means something different in C++ than it does when referring to an iterator. The latter only means that it cannot be used to update the database.

To configure the bulk retrieval buffer for an iterator when calling the `begin()` method, use the `BulkRetrievelItrOpt::bulk_retrieval(u_int32_t bulk_buffer_size)` function.

If you move a `db_vector_iterator` randomly rather than sequentially, then dbstl will not perform bulk retrieval because there is little performance gain from bulk retrieval in such an access pattern.

You can call `iterator::set_bulk_buffer()` to modify the iterator's bulk buffer size. Note that once bulk read is enabled, only the bulk buffer size can be modified. This means that bulk read

cannot be disabled. Also, if bulk read was not enabled when you created the iterator, you can't enable it after creation.

Example code using this feature can be found in the `TestAssoc::test_bulk_retrieval_read()` method, which is available in the the dbstl test suite.

Using the DB_RMW flag

The DB_RMW flag is an optimization for non-const (read-write) iterators. This flag causes the underlying cursor to acquire a write lock when reading so as to avoid deadlocks. Passing `ReadModifyWriteOption::read_modify_write()` to a container's `begin()` method creates an iterator whose cursor has this behavior.

Using secondary index database and secondary containers

Because duplicate keys are forbidden in primary databases, only `db_map`, `db_set` and `db_vector` are allowed to use primary databases. For this reason, they are called **primary containers**. A secondary database that supports duplicate keys can be used with `db_multimap` containers. These are called **secondary containers**. Finally, a secondary database that forbids duplicate keys can back a `db_map` container.

The **data_type** of this `db_multimap` secondary container is the **data_type** for the primary container. For example, a `db_map<int, Person>` object where the `Person` class has an `age` property of type `size_t`, a `db_multimap<size_t, Person>` using a secondary database allows access to a person by age.

A container created from a secondary database can only be used to iterate, search or delete. It can not be used to update or insert. While dbstl does expose the update and insert operations, Berkeley DB does not, and an exception will be thrown if attempts are made to insert objects into or update objects of a secondary container.

Example code demonstrating this feature is available in `TestAssoc::test_secondary_containers()`, which is available in the dbstl test suite.

Using transactions in dbstl

When using transactions with dbstl, you must call the dbstl transaction functions instead of the corresponding methods from the Berkeley DB C or C++ transaction API. That is, you must use `dbstl::begin_txn()`, `dbstl::commit_txn()` and `dbstl::abort_txn()` in order to begin/commit/abort transactions.

A container can be configured to use auto commit by setting the DB_AUTO_COMMIT flag when the environment or database handle is opened. In this case, any container method that supports auto commit will automatically form an independent transaction if the method is not in an external transactional context; Otherwise, the operation will become part of that transaction.

You can configure the flags used internally by dbstl when it is creating and committing these independent transactions required by auto commit. To do so, use the `db_container::set_txn_begin_flags()` and/or `db_container::set_commit_flags()` methods.

When a transaction is committed or aborted, dbstl will automatically close any cursors opened for use by the transaction. For this reason, any iterators opened within the transaction context should not be used after the transaction commits or aborts.

You can use nested transactions explicitly and externally, by calling `dbstl::begin_txn()` in a context already operating under the protection of a transaction. But you can not designate which transaction is the parent transaction. The parent transaction is automatically the most recently created and unresolved transaction in current thread.

It is also acceptable to use explicit transactions in a container configured for auto commit. The operation performed by the method will become part of the provided external transaction.

Finally, transactions and iterators cannot be shared among multiple threads. That is, they are not free-threaded, or thread-safe.

Using dbstl in multithreaded applications

Multithreaded use of dbstl must obey the following guidelines:

1. For a few non-standard platforms, you must first configure dbstl for that platform as described below. Usually the configure script will detect the applicable thread local storage (TLS) modifier to use, and then use it. If no appropriate TLS is found, the pthread TLS API is used.

   On HP Tru64, if you are not using a gcc compiler, #define HPTru64 before #include'ing any dbstl container header files.

2. Perform all initializations in a single thread. `dbstl::dbstl_startup()` should be called mutually exclusive in a single thread before using dbstl. If dbstl is used in only a single thread, this function does not need to be called.

   If necessary, callback functions for a complex type T must be registered to the singleton of DbstlElemTraits<T> before any container related to T (for example, `db_vector<T>`), is used, and certain isolation may be required among multiple threads. The best way to do this is to register all callback function pointers into the singleton in a single thread before making use of the containers.

   All container cursor open flags and auto commit transaction begin/commit flags must be set in a single thread before storing objects into or reading objects from the container.

3. Environment and database handles can optionally be shared across threads. If handles are shared, they must be registered in each thread that is using the handle (either directly, or indirectly using the containers that own the handles). You do this using the `dbstl::register_db()` and `dbstl::register_db_env()` functions. Note that these functions are not necessary if the current thread called `dbstl::open_db()` or `dbstl::open_env()` for the handle that is being shared. This is because the open functions automatically register the handle for you.

   Note that the get/set functions that provide access to container data members are not mutex-protected because these data members are supposed to be set only once at container

object initialization. Applications wishing to modify them after initialization must supply their own protection.

4. While container objects can be shared between multiple threads, iterators and transactions can not be shared.

5. Set the **directdb_get** parameter of the container `begin()` method to `true` in order to guarantee that referenced key/data pairs are always obtained from the database and not from an iterator's cached value. (This is the default behavior.) You should do this because otherwise a rare situation may occur. Given db_vector_iterator i1 and i2 used in the same iteration, setting *i1 = new_value will not update i2, and *i2 will return the original value.

6. If using a CDS database, only const iterators or read-only non-const iterators should be used for read only iterations. Otherwise, when multiple threads try to open read-write iterators at the same time, performance is greatly degraded because CDS only supports one write cursor open at any moment. The use of read-only iterators is good practice in general because dbstl contains internal optimizations for read-only iterators.

   To create a read-only iterator, do one of the following:

   - Use a `const` reference to the container object, then call the container's `begin()` method using the const reference, and then store the return value from the `begin()` method in a `db_vector::const_iterator`.

   - If you are using a non-const container object, then simply pass `true` to the **readonly** parameter of the non-const `begin()` method.

7. When using DS, CDS or TDS, enable the locking subsystem by passing the DB_INIT_LOCK flag to `DbEnv::open()`.

8. Perform portable thread synchronization within a process by calling the following functions. These are all global functions in the "dbstl" name space:

   ```
   db_mutex_t alloc_mutex();
   int lock_mutex(db_mutex_t);
   int unlock_mutex(db_mutex_t);
   void free_mutex(db_mutex_t);
   ```

   These functions use an internal dbstl environment's mutex functionality to synchronize. As a result, the synchronization is portable across all platforms supported by Berkeley DB.

The `WorkerThread` class provides example code demonstrating the use of dbstl in multi-threaded applications. You can find this class implemented in the dbstl test suite.

Working with primitive types

To store simple primitive types such as `int`, `long`, `double`, and so forth, an additional type parameter for the container class templates is needed. For example, to store an `int` in a `db_vector`, use this container class:

```
db_vector<int, ElementHolder<int> >;
```

To map integers to doubles, use this:

```
db_map<int, double, ElementHolder<double> >;
```

To store a `char*` string with `long` keys, use this:

```
db_map<long, char*, ElementHolder<char*> >;
```

Use this for `const char*` strings:

```
db_map<long, const char*, ElementHolder<const char*> >;
```

To map one const string to another, use this type:

```
db_map<const char*, const char*, ElementHolder<const char*> >;
```

The `TestVector::test_primitive()` method demonstrates more of these examples. You can find this method implemented in the dbstl test suite.

Storing strings

For `char*` and `wchar_t*` strings, `_DB_STL_StoreElement()` must be called following partial or total modifications before iterator movement, `container::operator[]` or `iterator::operator*/->` calls. Without the `_DB_STL_StoreElement()` call, the modified change will be lost. If storing an new value like this:

```
*iterator = new_char_star_string;
```

the call to `_DB_STL_StoreElement()` is not needed.

Note that passing a NULL pointer to a container of `char*` type or passing a `std::string` with no contents at all will insert an empty string of zero length into the database.

The string returned from a container will not live beyond the next iterator movement call, `container::operator[]` or `iterator::operator*/->` call.

A **db_map::value_type::second_type** or **db_map::datatype_wrap** should be used to hold a reference to a `container::operator[]` return value. Then the reference should be used for repeated references to that value. The *iterator is of type `ElementHolder<char *>`, which can be automatically converted to a `char *` pointer using its type conversion operator. Wherever an auto conversion is done by the compiler, the conversion operator of `ElementHolder<T>` is called. This avoids almost all explicit conversions, except for two use cases:

1. The *iterator is used as a "..." parameter like this:

   ```
   printf("this is the special case %s", *iterator);
   ```

   This compiles but causes errors. Instead, an explicit cast should be used:

   ```
   printf("this is the special case %s", (char *)*iterator);
   ```

2. For some old compilers, such as gcc3.4.6, the *iterator cannot be used with the ternary ? operator, like this:

```
expr ? *iterator : var
```

Even when **var** is the same type as the iterator's `value_type`, the compiler fails to perform an auto conversion.

When using `std::string` or `std::wstring` as the data type for dbstl containers — that is, `db_vector<string>`, and `db_map<string, wstring>` — the string's content rather than the string object itself is stored in order to maintain persistence.

You can find example code demonstrating string storage in the `TestAssoc::test_char_star_string_storage()` and `TestAssoc::test_storing_std_strings()` methods. These are available in the dbstl test suite.

Store and Retrieve data or objects of complex types

Storing varying length objects

A structure like this:

```
class SMSMsg
{
public:
 size_t mysize;
 time_t when;
 size_t szmsg;
 int to;
 char msg[1];
};
```

with a varying length string in `msg` cannot simply be stored in a `db_vector<SMSMsg>` without some configuration on your part. This is because, by default, dbstl uses the **sizeof()** operator to get the size of an object and then `memcpy()` to copy the object. This process is not suitable for this use-case as it will fail to capture the variable length string contained in `msg`.

There are currently two ways to store these kind of objects:

1. Register callback functions with dbstl that are used to measure an object's size, and then marshal/unmarshal the object.

2. Use a `DbstlDbt` wrapper object.

Storing by marshaling objects

One way to store an object that contains variable-sized fields is to marshall all of the object's data into a single contiguous area in memory, and then store the contents of that buffer. This means that upon retrieval, the contents of the buffer must be unmarshalled. To do these things, you must register three callback functions:

- `typedef void (*ElemRstoreFunct)(T& dest, const void *srcdata);`

This callback is used to unmarshal an object, updating **dest** using data found in **srcdata**. The data in **srcdata** contains the chunk of memory into which the object was originally marshalled. The default unmarshalling function simply performs a cast (for example, `dest = *((T*)srcdata)`), which assumes the **srcdata** simply points to the memory layout of the object.

- `typedef size_t (*ElemSizeFunct)(const T& elem);`

  This callback returns the size in bytes needed to store the **elem** object. By default this function simply uses **sizeof(elem)** to determine the size of **elem**.

- `typedef void (*ElemCopyFunct)(void *dest, const T&elem);`

  This callback is used to arrange all data contained by **elem** into the chunk of memory to which **dest** refers. The size of **dest** is set by the `ElemSizeFunct` function, discussed above. The default marshalling function simply uses `memcpy()` to copy **elem** to **dest**.

The `DbstlElemTraits<SMSMsg>::instance()->set_size_function()`, `set_copy_function()` and `set_restore_function()` methods are used to register these callback functions. If a callback is not registered, its default function is used.

By providing non-default implementations of the callbacks described here, you can store objects of varying length and/or objects which do not reside in a continuous memory chunk — for example, objects containing a pointer which refers another object, or a string, and so forth. As a result, containers/iterators can manage variable length objects in the same as they would manage objects that reside in continuous chunks of memory and are of identical size.

Using a `DbstlDbt` wrapper object

To use a `DbstlDbt` wrapper object to store objects of variable length, a `db_vector<DbstlDbt>` container is used to store complex objects in a `db_vector`. `DbstlDbt` derives from DB C++ API's `Dbt`class, but can manage its referenced memory properly and release it upon destruction. The memory referenced by `DbstlDbt` objects is required to be allocated using the `malloc()/realloc()` functions from the standard C library.

Note that the use of `DbstlDbt` wrapper class is not ideal. It exists only to allow raw bytes of no specific type to be stored in a container.

To store an `SMSMsg` object into a `db_vector<DbstlDbt>` container using a `DbstlDbt` object:

1. Wrap the `SMSMSg` object into a `DbstlDbt` object, then marshal the SMSMsg object properly into the memory chunk referenced by `DbstlDbt::data`.

2. Store the `DbstlDbt` object into a `db_vector<DbstlDbt>` container. The bytes in the memory chunk referenced by the `DbstlDbt` object's **data** member are stored in the `db_vector<DbstlDbt>` container.

3. Reading from the container returns a `DbstlDbt` object whose **data** field points to the `SMSMsg` object located in a continuous chunk of memory. The application needs to perform its own unmarshalling.

4. The memory referenced by DbstlDbt::data is freed automatically, and so the application should not attempt to free the memory.

ElementHolder should not be used to store objects of a class because it doesn't support access to object members using **(*iter).member** or **iter->member** expressions. In this case, the default ElementRef<ddt> is used automatically.

ElementRef inherits from ddt, which allows ***iter** to return the object stored in the container. (Technically it is an ElementRef<ddt> object, whose "base class" part is the object you stored). There are a few data members and member functions in ElementRef, which all start with _DB_STL_. To avoid potential name clashes, applications should not use names prefixing _DB_STL_ in classes whose instances may be stored into dbstl containers.

Example code demonstrating this feature can be found in the TestAssoc::test_arbitrary_object_storage method, which can be located in the dbstl test suite.

Storing arbitrary sequences

A sequence is a group of related objects, such as an array, a string, and so forth. You can store sequences of any structure using dbstl, so long as you implement and register the proper callback functions. By using these callbacks, each object in the sequence can be a complex object with data members that are all not stored in a continuous memory chunk.

Note that when using these callbacks, when you retrieve a stored sequence from the database, the entire sequence will reside in a single continuous block of memory with the same layout as that constructed by your sequence copy function.

For example, given a type RGB:

```
struct RGB{char r, g, b, bright;};
```

and an array of RGB objects, the following steps describe how to store an array into one key/data pair of a db_map container.

1. Use a db_map<int, RGB *, ElementHolder<RGB *> > container.

2. Define two functions. The first returns the number of objects in a sequence, the second that copies objects from a sequence to a defined destination in memory:

   ```
   typedef size_t (*SequenceLenFunct)(const RGB*);
   ```

   and

   ```
   typedef void (*SequenceCopyFunct)(RGB*dest, const RGB*src);
   ```

3. Call DbstlElemTraits<RGB>::set_sequence_len_function()/set_sequence_copy_function() to register them as callbacks.

The `SequenceLenFunct` function

```
typedef size_t (*SequenceLenFunct)(const RGB*);
```

A `SequenceLenFunct` function returns the number of objects in a sequence. It is called when inserting into or reading from the database, so there must be enough information in the sequence itself to enable the `SequenceLenFunct` function to tell how many objects the sequence contains. The `char*` and `wchar_t*` strings use a `'\0'` special character to do this. For example, RGB(0, 0, 0, 0) could be used to denote the end of the sequence. Note that for your implementation of this callback, you are not required to use a trailing object with a special value like `'\0'` or `RGB(0, 0, 0, 0)` to denote the end of the sequence. You are free to use what mechanism you want in your `SequenceLenFunct` function implementation to figure out the length of the sequence.

The `SequenceCopyFunct` function

```
typedef void (*SequenceCopyFunct)(RGB*dest, const RGB*src);
```

`SequenceCopyFunct` copies objects from the sequence **src** into memory chunk **dest**. If the objects in the sequence do not reside in a continuous memory chunk, this function must marshal each object in the sequence into the **dest** memory chunk.

The sequence objects will reside in the continuous memory chunk referred to by **dest**, which has been sized by `SequenceLenFunct` and `ElemSizeFunct` if available (which is when objects in the sequence are of varying lengths). `ElemSizeFunct` function is not needed in this example because **RGB** is a simple fixed length type, the `sizeof()` operator is sufficient to return the size of the sequence.

Notes

- The get and set functions of this class are not protected by any mutexes. When using multiple threads to access the function pointers, the callback functions must be registered to the singleton of this class before any retrieval of the callback function pointers. Isolation may also be required among multiple threads. The best way is to register all callback function pointers in a single thread before making use of the any containers.
- If objects in a sequence are not of identical sizes, or are not located in a consecutive chunk of memory, you also need to implement and register the `DbstlElemTraits<>::ElemSizeFunct` callback function to measure the size of each object. When this function is registered, it is also used when allocating memory space.

  There is example code demonstrating the use this feature in `TestAssoc::test_arbitray_sequence_storage()`, which is available in the dbstl test suite.
- A consequence of this dbstl feature is that you can not store a pointer value directly because dbstl will think it is a sequence head pointer. Instead, you need to convert the pointer into a `long` and then store it into a `long` container. And please note that pointer values are probably meaningless if the stored value is to be used across different application run times.

Dbstl persistence

The following sections provide information on how to achieve persistence using dbstl.

Direct database get

Each container has a **begin()** method which produces an iterator. These **begin** methods take a boolean parameter, **directdb_get**, which controls the caching behavior of the iterator. The default value of this parameter is `true`.

If **directdb_get** is `true`, then the persistent object is fetched anew from the database each time the iterator is dereferenced as a pointer by use of the star-operator (***iterator**) or by use of the arrow-operator (**iterator->member**). If **directdb_get** is `false`, then the first dereferencing of the iterator fetches the object from the database, but later dereferences can return cached data.

With **directdb_get** set to `true`, if you call:

```
(*iterator).datamember1=new-value1;
(*iterator).datamember2=new-value2;
```

then the assignment to `datamember1` will be lost, because the second dereferencing of the iterator would cause the cached copy of the object to be overwritten by the object's persistent data from the database.

You also can use the arrow operator like this:

```
iterator->datamember1=new-value1;
iterator->datamember2=new-value2;
```

This works exactly the same way as **iterator::operator***. For this reason, the same caching rules apply to arrow operators as they do for star operators.

One way to avoid this problem is to create a reference to the object, and use it to access the object:

```
container::value_type &ref = *iterator;
ref.datamember1=new-value1;
ref.datamember2=new-value2;
...// more member function calls and datamember assignments
ref._DB_STL_StoreElement();
```

The above code will not lose the newly assigned value of `ref.datamember1` in the way that the previous example did.

In order to avoid these complications, you can assign to the object referenced by an iterator with another object of the same type like this:

```
container::value_type obj2;
obj2.datamember1 = new-value1;
obj2.datamember2 = new-value2;
*itr = obj2;
```

This code snippet causes the new values in `obj2` to be stored into the underlying database.

If you have two iterators going through the same container like this:

```
for (iterator1 = v.begin(), iterator2 = v.begin();
     iterator1 != v.end();
     ++iterator1, ++iterator2) {
        *iterator1 = new_value;
        print(*iterator2);
}
```

then the printed value will depend on the value of **directdb_get** with which the iterator had been created. If **directdb_get** is `false`, then the original, persistent value is printed; otherwise the newly assigned value is returned from the cache when `iterator2` is dereferenced. This happens because each iterator has its own cached copy of the persistent object, and the dereferencing of `iterator2` refreshes `iterator2`'s copy from the database, retrieving the value stored by the assignment to `*iterator1`.

Alternatively, you can set **directdb_get** to `false` and call `iterator2->refresh()` immediately before the dereferencing of `iterator2`, so that `iterator2`'s cached value is refreshed.

If **directdb_get** is `false`, a few of the tests in dbstl's test kit will fail. This is because the above contrived case appears in several of C++ STL tests. Consequently, the default value of the **directdb_get** parameter in the `container::begin()` methods is `true`. If your use cases avoid such bizarre usage of iterators, you can set it to `false`, which makes the iterator read operation faster.

Change persistence

If you modify the object to which an iterator refers by using one of the following:

```
(*iterator).member_function_call()
```

or

```
(*iterator).data_member = new_value
```

then you should call `iterator->_DB_STL_StoreElement()` to store the change. Otherwise the change is lost after the iterator moves on to other elements.

If you are storing a sequence, and you modified some part of it, you should also call `iterator->_DB_STL_StoreElement()` before moving the iterator.

And in both cases, if **directdb_get** is `true` (this is the default value), you should call `_DB_STL_StoreElement()` after the change and before the next iterator movement OR the next dereferencing of the iterator by the star or arrow operators (`iterator::operator*` or `iterator::operator->`). Otherwise, you will lose the change.

If you update the element by assigning to a dereferenced iterator like this:

```
*iterator = new_element;
```

then you never have to call `_DB_STL_StoreElement()` because the change is stored in the database automatically.

Object life time and persistence

Dbstl is an interface to Berkeley DB, so it is used to store data persistently. This is really a different purpose from that of regular C++ STL. This difference in their goals has implications on expected object lifetime: In standard STL, when you store an object A of type ID into C++ stl vector V using V.push_back(A), if a proper copy constructor is provided in A's class type, then the copy of A (call it B) and everything in B, such as another object C pointed to by B's data member B.c_ptr, will be stored in V and will live as long as B is still in V and V is alive. B will be destroyed when V is destroyed or B is erased from V.

This is not true for dbstl, which will copy A's data and store it in the underlying database. The copy is by default a shallow copy, but users can register their object marshalling and unmarshalling functions using the `DbstlElemTraits` class template. So if A is passed to a `db_vector` container, `dv`, by using `dv.push_back(A)`, then dbstl copies A's data using the registered functions, and stores data into the underlying database. Consequently, A will be valid, even if the container is destroyed, because it is stored into the database.

If the copy is simply a shallow copy, and A is later destroyed, then the pointer stored in the database will become invalid. The next time we use the retrieved object, we will be using an invalid pointer, which probably will result in errors. To avoid this, store the referred object C rather than the pointer member A.c_ptr itself, by registering the right marshalling/unmarshalling function with `DbstlElemTraits`.

For example, consider the following example class declaration:

```
class ID
{
public:
    string Name;
    int Score;
};
```

Here, the class ID has a data member **Name**, which refers to a memory address of the actual characters in the string. If we simply shallow copy an object, `id`, of class ID to store it, then the stored data, `idd`, is invalid when `id` is destroyed. This is because `idd` and `id` refer to a common memory address which is the base address of the memory space storing all characters in the string, and this memory space is released when `id` is destroyed. So `idd` will be referring to an invalid address. The next time we retrieve `idd` and use it, there will probably be memory corruption.

The way to store `id` is to write a marshal/unmarshal function pair like this:

```
void copy_id(void *dest, const ID&elem)
{
 memcpy(dest, &elem.Score, sizeof(elem.Score));
 char *p = ((char *)dest) + sizeof(elem.Score);
 strcpy(p, elem.Name.c_str());
```

```
}

void restore_id(ID& dest, const void *srcdata)
{
 memcpy(&dest.Score, srcdata, sizeof(dest.Score));
 const char *p = ((char *)srcdata) + sizeof(dest.Score);
 dest.Name = p;
}

size_t size_id(const ID& elem)
{
 return sizeof(elem.Score) + elem.Name.size() +
     1;// store the '\0' char.
}
```

Then register the above functions before storing any instance of `ID`:

```
DbstlElemTraits<ID>::instance()->set_copy_function(copy_id);
DbstlElemTraits<ID>::instance()->set_size_function(size_id);
DbstlElemTraits<ID>::instance()->set_restore_function(restore_id);
```

This way, the actual data of instances of ID are stored, and so the data will persist even if the container itself is destroyed.

Dbstl container specific notes

db_vector specific notes

- Set the DB_RENUMBER flag in the database handle if you want `db_vector<>` to work like `std::vector` or `std::deque`. Do not set DB_RENUMBER if you want `db_vector<>` to work like `std::list`. Note that without DB_RENUMBER set, `db_vector<>` can work faster.

  For example, to construct a fast std::queue/std::stack object, you only need a `db_vector<>` object whose database handle does not have DB_RENUMBER set. Of course, if the database handle has DB_RENUMBER set, it still works for this kind of scenario, just not as fast.

  `db_vector` does not check whether DB_RENUMBER is set. If you do not set it, `db_vector<>` will not work like std::vector<>/std::deque<> with regard to operator[], because the indices are not maintained in that case.

  You can find example code showing how to use this feature in the `TestVector::test_queue_stack()` method, which is available in the dbstl test suite.

- Just as is the case with `std::vector`, inserting/deleting in the middle of a `db_vector` is slower than doing the same action at the end of the sequence. This is because the underlying DB_RECNO DB (with the DB_RENUMBER flag set) is relatively slow when inserting/deleting in the middle or the head — it has to update the index numbers of all the records following the one that was inserted/deleted. If you do not need to keep the index ordered on insert/delete, you can use `db_map` instead.

db_vector also contains methods inherited from std::list and std::deque, including std::list<>'s unique methods remove(), remove_if(), unique(), merge(), sort(), reverse(), and splice(). These use the identical semantics/behaviors of the std::list<> methods, although pushing/deleting at the head is slower than the std::deque and std::list equivalent when there are quite a lot of elements in the database.

- You can use std::queue, std::priority_queue and std::stack container adapters with db_vector; they work with db_vector even without DB_RENUMBER set.

Associative container specific notes

db_map contains the union of method set from std::map and hash_map, but there are some methods that can only be called on containers backed by DB_BTREE or DB_HASH databases. You can call db_map<>::is_hash() to figure out the type of the backing database. If you call unsupported methods then an InvalidFunctionCall exception is thrown.

These are the DB_BTREE specific methods: upper_bound(), lower_bound(), key_comp(), and value_comp(). The DB_HASH specific methods are key_eq(), hash_funct().

Using dbstl efficiently

Using iterators efficiently

To make the most efficient possible use of iterators:

- Close an iterator's cursor as soon as possible.

  Each iterator has an open cursor associated with it, so when you are finished using the iterator it is a good habit to explicitly close its cursor. This can potentially improve performance by avoiding locking issues, which will enhanced concurrency. Dbstl will close the cursor when the iterator is destroyed, but you can close the cursor before that time. If the cursor is closed, the associated iterator cannot any longer be used.

  In some functions of container classes, an iterator is used to access the database, and its cursor is internally created by dbstl. So if you want to specify a non-zero flag for the Db::cursor() call, you need to call the container's set_cursor_open_flag() function to do so.

- Use const iterators where applicable.

  If your data access is read only, you are strongly recommended to use a const iterator. In order to create a const iterator, you must use a const reference to the container object. For example, supposed we have:

  ```
  db_vector<int> intv(10);
  ```

  then we must use a:

  ```
  const db_vector<int>& intv_ref = intv;
  ```

reference to invoke the const begin/end functions. `intv_ref.begin()` will give you a const iterator. You can use a const iterator only to read its referenced data elements, not update them. However, you should have better performance with this iterator using, for example, either `iterator::operator*` or `iterator::operator->member`. Also, using array indices like `intv_ref[i]` will also perform better.

All functions in dbstl's containers which return an iterator or data element reference have two versions — one returns a const iterator/reference, the other returns an iterator/reference. If your access is read only, choose the version returning const iterators/references.

Remember that you can only use a const reference to a container object to call the const versions of `operator*` and `operator[]`.

You can also use the non-const container object or its non-const reference to create a read only iterator by passing `true` to the **readonly** parameter in the container's `begin()` method.

- Use pre-increment/pre-decrement rather than post-increment/post-decrement where possible

  Pre-increment operations are more efficient because the `++iterator` avoids two iterator copy constructions. This is true when you are using C++ standard STL iterators as well.

- Use bulk retrieval in iterators

  If your access pattern is to go through the entire database read only, or if you are reading a continuous range of the database, bulk retrieval can be very useful because it returns multiple key/data pairs in one database call. But be aware that you can only read the returned data, you can not update it. Also, if you do a bulk retrieval and read the data, and simultaneously some other thread of control updates that same data, then unless you are using a serializable transaction, you will now be working with old data.

Using containers efficiently

To make the most efficient possible use of containers:

- Avoid using container methods that return references. These because they are a little more expensive.

  To implement reference semantics, dbstl has to wrap the data element with the current key/data pair, and must invoke two iterator copy constructions and two Berkeley DB cursor duplications for each such a call. This is true of non-const versions of these functions:

  ```
  db_vector<T>::operator[]()
  db_vector<T>::front()
  db_vector<T>::back()
  db_vector<T>::at()
  db_map<>::operator[]()
  ```

  There are alternatives to these functions, mainly through explicit use of iterators.

- Use const containers where possible.

The const versions of the functions listed above have less overhead than their non-const counterparts. Using const containers and iterators can bring more performance when you call the const version of the overloaded container/iterator methods. To do so, you define a const container reference to an existing container, and then use this reference to call the methods. For example, if you have:

```
db_vector<int> container int_vec
```

then you can define a const reference to `int_vec`:

```
const db_vector<int>& int_vec_ref;
```

Then you use `int_vec_ref.begin()` to create a const iterator, `citr`. You can now can use `int_vec_ref` to call the const versions of the container's member functions, and then use `citr` to access the data read only. By using `int_vec_ref` and `citr`, we can gain better performance.

It is acceptable to call the non-const versions of container functions that return non-const iterators, and then assign these return values to const iterator objects. But if you are using Berkeley DB concurrent data store (CDS), be sure to set the **readonly** parameter for each container method that returns an iterator to `true`. This is because each iterator corresponds to a Berkeley DB cursor, and so for best performance you should specify that the returned iterator be read-only so that the underlying cursor is also read-only. Otherwise, the cursor will be a writable cursor, and performance might be somewhat degraded. If you are not using CDS, but instead TDS or DS or HA, there is no distinction between read-only cursors and read-write cursors. Consequently, you do not need to specify the **readonly** parameter at all.

Dbstl memory management

Freeing memory

When using dbstl, make sure memory allocated in the heap is released after use. The rules for this are:

- dbstl will free/delete any memory allocated by dbstl itself.
- You are responsible for freeing/deleting any memory allocated by your code outside of dbstl.

Type specific notes

DbEnv/Db

When you open a `DbEnv` or `Db` object using `dbstl::open_env()` or `dbstl::open_db()`, you do not need to delete that object. However, if you new'd that object and then opened it without using the `dbstl::open_env()` or `dbstl::open_db()` methods, you are responsible for deleting the object.

Note that you must `new` the `Db` or `DbEnv` object, which allocates it on the heap. You can not allocate it on the stack. If you do, the order of destruction is uncontrollable, which makes dbstl unable to work properly.

You can call dbstl_exit() before the process exits, to release any memory allocated by dbstl that has to live during the entire process lifetime. Releasing the memory explicitly will not make much difference, because the process is about to exit and so all memory allocated on the heap is going to be returned to the operating system anyway. The only real difference is that your memory leak checker will not report false memory leaks.

dbstl_exit() releases any memory allocated by dbstl on the heap. It also performs other required shutdown operations, such as closing any databases and environments registered to dbstl and shared across the process.

If you are calling the dbstl_exit() function, and your DbEnv or Db objects are new'd by your code, the dbstl_exit() function should be called before deleting the DbEnv or Db objects, because they need to be closed before being deleted. Alternatively, you can call the dbstl::close_env() or dbstl::close_db() functions before deleting the DbEnv or Db objects in order to explicitly close the databases or environments. If you do this, can then delete these objects, and then call dbstl_exit().

DbstlDbt

Only when you are storing raw bytes (such as a bitmap) do you have to store and retrieve data by using the DbstlDbt helper class. Although you also can do so simply by using the Berkeley DB Dbt class, the DbstlDbt class offers more convenient memory management behavior.

When you are storing DbstlDbt objects (such as db_vector<DbstlDbt>), you *must* allocate heap memory explicitly using the malloc() function for the DbstlDbt object to reference, but you do not need to free the memory – it is automatically freed by the DbstlDbt object that owns it by calling the standard C library free() function.

However, because dbstl supports storing any type of object or primitive data, it is rare that you would have to store data using DbstlDbt objects while using dbstl. Examples of storing DbstlDbt objects can be found in the TestAssoc::test_arbitrary_object_storage() and TestAssoc::test_char_star_string_storage() functions, which are available in the dbstl test suite.

Dbstl miscellaneous notes

Special notes about trivial methods

There are some standard STL methods which are meaningless in dbstl, but they are kept in dbstl as no-ops so as to stay consistent with the standard. These are:

```
db_vecter::reserve();
db_vector::max_size();
db_vector::capacity();
db_map::reserve();
db_map::max_size();
```

db_vector<>::max_size() and db_map<>::max_size() both return 2^30. This does not mean that Berkeley DB can only hold that much data. This value is returned to conform to some

compilers' overflow rules — if we set bigger numbers like 2^32 or 2^31, some compilers complain that the number has overflowed.

See the Berkeley DB documentation for information about limitations on how much data a database can store.

There are also some read-only functions. You set the configuration for these using the Berkeley DB API. You access them using the container's methods. Again, this is to keep consistent with C++ standard STL containers, such as:

```
db_map::key_comp();
db_map::value_comp();
db_map::hash_funct();
db_map::key_eq();
```

Using correct container and iterator public types

All public types defined by the C++ STL specification are present in dbstl. One thing to note is the **value_type**. dbstl defines the **value_type** for each iterator and container class to be the raw type without the `ElementRef/ElementHolder` wrapper, so this type of variable can not be used to store data in a database. There is a **value_type_wrap** type for each container and iterator type, with the raw type wrapped by the `ElementRef/ElementHolder`.

For example, when type `int_vector_t` is defined as

```
db_vector<int, ElementHolder<int> >
```

its **value_type** is `int`, its **value_type_wrap** is `ElementHolder<int>`, and its reference and pointer types are `ElementHolder<int>&` and `ElementHolder<int>*` respectively. If you need to store data, use **value_type_wrap** to make use of the wrapper to store data into database.

The reason we leave **value_type** as the raw type is that we want the existing algorithms in the STL library to work with dbstl because we have seen that without doing so, a few tests will fail.

You need to use the same type as the return type of the data element retrieval functions to hold a value in order to properly manipulate the data element. For example, when calling

```
db_vector<T>::operator[]
```

check that the return type for this function is

```
db_vector<T>::datatype_wrap
```

Then, hold the return value using an object of the same type:

```
db_vector<T>::datatype_wrap refelem = vctr[3];
```

Dbstl known issues

Three algorithm functions of gcc's C++ STL test suite do not work with dbstl. They are `find_end()`, `inplace_merge()` and `stable_sort()`.

The reason for the incompatibility of `find_end()` is that it assumes the data an iterator refers to is located at a shared place (owned by its container). This assumption is not correct in that it is part of the C++ STL standards specification. However, this assumption can not be true for dbstl because each dbstl container iterator caches its referenced value.

Consequently, please do not use `find_end()` for dbstl container iterators if you are using gcc's STL library.

The reason for the incompatibility with `inplace_merge()` and `stable_sort()` is that their implementation in gcc requires the **value_type** for a container to be default constructible. This requirement is not a part of the the C++ STL standard specification. Dbstl's value type wrappers (such as `ElementHolder`) do not support it.

These issues do not exist for any function available with the Microsoft Visual C++ STL library.

Chapter 8. Berkeley DB Architecture

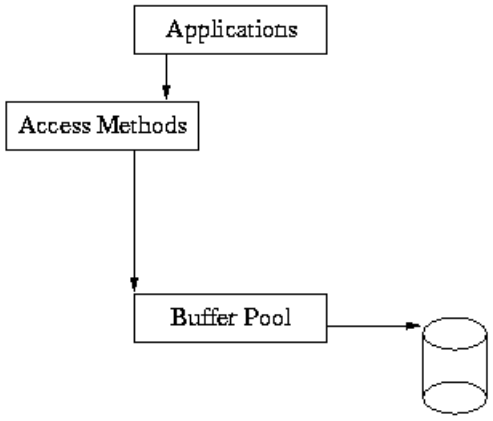
The big picture

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() and finished with DB_TXN->commit(). If the operation fails due to a deadlock, the transaction is aborted using DB_TXN->abort(), 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() and DB->cursor() 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.

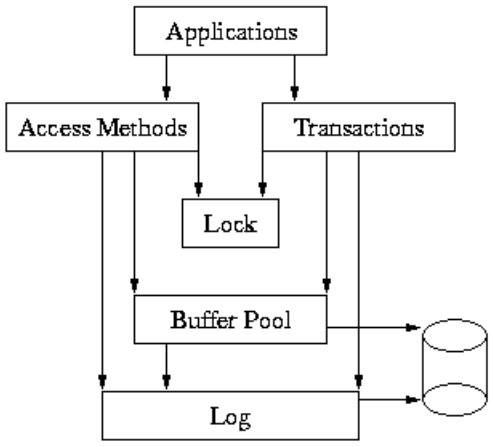
Locking

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. 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.

Programmatic APIs

The Berkeley DB subsystems can be accessed through interfaces from multiple languages. Applications can use Berkeley DB via C, C++ or Java, as well as a variety of scripting languages such as Perl, Python, Ruby or Tcl. Environments can be shared among applications written by using any of these interfaces. 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.

STL

dbstl is an C++ STL style API for Berkeley DB, based on the C++ API above. With it, you can store data/objects of any type into or retrieve them from Berkeley DB databases as if you are using C++ STL containers. The full functionality of Berkeley DB can still be utilized via dbstl with little performance overhead, e.g. you can use all transaction and/or replication functionality of Berkeley DB.

dbstl container/iterator class templates reside in header files dbstl_vector.h, dbstl_map.h and dbstl_set.h. Among them, dbstl_vector.h contains dbstl::db_vector and its iterators; dbstl_map.h contains dbstl::db_map, dbstl::db_multimap and their iterators; dbstl_set.h contains dbstl::db_set and dbstl::db_multiset and their iterators. You should include needed header file(s) to use the container/iterator. Note that we don't use the file name with no extention --- To use dbstl::db_vector, you should do this:

```
#include "dbstl_vector.h"
```

rather than this:

```
#include "dbstl_vector"
```

And these header files reside in "stl" directory inside Berkeley DB source root directory. If you have installed Berkeley DB, they are also available in the "include" directory in the directory where Berkeley DB is installed.

Apart from the above three header files, you may also need to include db_exception.h and db_utility.h files. The db_exception.h file contains all exception classes of dbstl, which integrate seamlessly with Berkeley DB C++ API exceptions and C++ standard exception classes in std namespace. And the db_utility.h file contains the DbstlElemTraits which helps you to store complex objects. These five header files are all that you need to include in order to make use of dbstl.

All symbols of dbstl, including classes, class templates, global functions, etc, reside in the namespace "dbstl", so in order to use them, you may also want to do this:

```
using namespace dbstl;
```

The dbstl library is always at the same place where Berkeley DB library is located, you will need to build it and link with it to use dbstl.

While making use of dbstl, you will probably want to create environment or databases directly, or set/get configurations to Berkeley DB environment or databases, etc. You are allowed to do so via Berkeley DB C/C++ API.

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 dbm and hsearch interfaces. 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 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 wrappers 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. U Be

db_deadlock utility

The db_deadlock 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 utility

The db_dump utility writes a copy of the database to a flat-text file in a portable format.

db_hotbackup utility

The db_hotbackup utility creates "hot backup" or "hot failover" snapshots of Berkeley DB database environments.

db_load utility

The db_load utility reads the flat-text file produced by the db_load utility and loads it into a database file.

db_printlog utility

The db_printlog utility displays the contents of Berkeley DB log files in a human-readable and parsable format.

db_recover utility

The db_recover 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_sql utility

The db_sql utility translates a schema description written in a SQL Data Definition Language dialect into C code that implements the schema using Berkeley DB.

db_stat utility

The db_stat utility displays statistics for databases and database environments.

db_upgrade utility

The db_upgrade utility provides a command-line interface for upgrading underlying database formats.

db_verify utility

The db_verify 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.

Chapter 9. The Berkeley DB Environment

Database environment introduction

A Berkeley DB environment is an encapsulation of one or more databases, log files and region files. Region files are the shared memory areas that contain information about the database environment such as memory pool cache pages. Only databases are byte-order independent and only database files can be moved between machines of different byte orders. Log files can be moved between machines of the same byte order. Region files are usually unique to a specific machine and potentially to a specific operating system release.

The simplest way to administer a Berkeley DB application environment is to create a single **home** directory that stores the files for the applications that will share the environment. The environment home directory must be created before any Berkeley DB applications are run. Berkeley DB itself never creates the environment home directory. The environment can then be identified by the name of that directory.

An environment may be shared by any number of processes, as well as by any number of threads within those processes. It is possible for an environment to include resources from other directories on the system, and applications often choose to distribute resources to other directories or disks for performance or other reasons. However, by default, the databases, shared regions (the locking, logging, memory pool, and transaction shared memory areas) and log files will be stored in a single directory hierarchy.

It is important to realize that all applications sharing a database environment implicitly trust each other. They have access to each other's data as it resides in the shared regions, and they will share resources such as buffer space and locks. At the same time, any applications using the same databases **must** share an environment if consistency is to be maintained between them.

Database Environment Operations	Description
db_env_create()	Create an environment handle
DB->getenv() handle	Return DB's underlying DB_ENV handle
DB_ENV->close()	Close an environment
DB_ENV->dbremove()	Remove a database
DB_ENV->dbrename()	Rename a database
DB_ENV->err()	Error message
DB_ENV->failchk()	Check for thread failure
DB_ENV->fileid_reset()	Reset database file IDs
DB_ENV->open()	Return environment's home directory
DB_ENV->open()	Return flags with which the environment was opened
DB_ENV->lsn_reset()	Reset database file LSNs
DB_ENV->open()	Open an environment

Database Environment Operations	Description
DB_ENV->remove()	Remove an environment
DB_ENV->stat()	Environment statistics
db_strerror()	Error strings
DB_ENV->version()	Return version information
Environment Configuration	
DB_ENV->set_alloc()	Set local space allocation functions
DB_ENV->set_app_dispatch()	Configure application recovery
DB_ENV->set_cachesize()	Set the environment cache size
DB_ENV->set_data_dir()	Set the environment data directory
DB_ENV->set_encrypt()	Set the environment cryptographic key
DB_ENV->set_errcall()	Set error and informational message callbacks
DB_ENV->set_errfile()	Set error and informational message FILE
DB_ENV->set_errpfx()	Set error message prefix
DB_ENV->set_event_notify()	Set event notification callback
DB_ENV->set_feedback()	Set feedback callback
DB_ENV->set_flags()	Environment configuration
DB_ENV->set_isalive()	Set thread is-alive callback
DB_ENV->set_intermediate_dir_mode()	Set intermediate directory creation mode
DB_ENV->set_shm_key()	Set system memory shared segment ID
DB_ENV->set_thread_id()	Set thread of control ID function
DB_ENV->set_thread_count()	Set approximate thread count
DB_ENV->set_thread_id_string()	Set thread of control ID format function
DB_ENV->set_timeout()	Set lock and transaction timeout
DB_ENV->set_tmp_dir()	Set the environment temporary file directory
DB_ENV->set_verbose()	Set verbose messages

Creating a database environment

The Berkeley DB environment is created and described by the db_env_create() and DB_ENV->open() interfaces. In situations where customization is desired, such as storing log files on a separate disk drive or selection of a particular cache size, applications must describe the customization by either creating an environment configuration file in the environment home directory or by arguments passed to other DB_ENV handle methods.

Once an environment has been created, database files specified using relative pathnames will be named relative to the home directory. Using pathnames relative to the home directory allows the entire environment to be easily moved, simplifying restoration and recovery of a database in a different directory or on a different system.

Applications first obtain an environment handle using the db_env_create() method, then call the DB_ENV->open() method which creates or joins the database environment. There are a number of options you can set to customize DB_ENV->open() for your environment. These options fall into four broad categories:

Subsystem Initialization:

These flags indicate which Berkeley DB subsystems will be initialized for the environment, and what operations will happen automatically when databases are accessed within the environment. The flags include DB_INIT_CDB, DB_INIT_LOCK, DB_INIT_LOG, DB_INIT_MPOOL, and DB_INIT_TXN. The DB_INIT_CDB flag does initialization for Berkeley DB Concurrent Data Store applications. (See Concurrent Data Store introduction (page 136) for more information.) The rest of the flags initialize a single subsystem; that is, when DB_INIT_LOCK is specified, applications reading and writing databases opened in this environment will be using locking to ensure that they do not overwrite each other's changes.

Recovery options:

These flags, which include DB_RECOVER and DB_RECOVER_FATAL, indicate what recovery is to be performed on the environment before it is opened for normal use.

Naming options:

These flags, which include DB_USE_ENVIRON and DB_USE_ENVIRON_ROOT, modify how file naming happens in the environment.

Miscellaneous:

Finally, there are a number of miscellaneous flags, for example, DB_CREATE which causes underlying files to be created as necessary. See the DB_ENV->open() manual pages for further information.

Most applications either specify only the DB_INIT_MPOOL flag or they specify all four subsystem initialization flags (DB_INIT_MPOOL, DB_INIT_LOCK, DB_INIT_LOG, and DB_INIT_TXN). The former configuration is for applications that simply want to use the basic Access Method interfaces with a shared underlying buffer pool, but don't care about recoverability after application or system failure. The latter is for applications that need recoverability. There are situations in which other combinations of the initialization flags make sense, but they are rare.

The DB_RECOVER flag is specified by applications that want to perform any necessary database recovery when they start running. That is, if there was a system or application failure the last time they ran, they want the databases to be made consistent before they start running again. It is not an error to specify this flag when no recovery needs to be done.

The DB_RECOVER_FATAL flag is more special-purpose. It performs catastrophic database recovery, and normally requires that some initial arrangements be made; that is, archived log files be brought back into the filesystem. Applications should not normally specify this flag. Instead, under these rare conditions, the db_recover utility should be used.

The following is a simple example of a function that opens a database environment for a transactional program.

```
DB_ENV *
db_setup(home, data_dir, errfp, progname)
```

```
 char *home, *data_dir, *progname;
 FILE *errfp;
{
 DB_ENV *dbenv;
 int ret;

 /*
  * Create an environment and initialize it for additional error
  * reporting.
  */
 if ((ret = db_env_create(&dbenv, 0)) != 0) {
  fprintf(errfp, "%s: %s\n", progname, db_strerror(ret));
  return (NULL);
 }
 dbenv->set_errfile(dbenv, errfp);
 dbenv->set_errpfx(dbenv, progname);

 /*
  * Specify the shared memory buffer pool cachesize: 5MB.
  * Databases are in a subdirectory of the environment home.
  */
 if ((ret = dbenv->set_cachesize(dbenv, 0, 5 * 1024 * 1024, 0)) != 0) {
  dbenv->err(dbenv, ret, "set_cachesize");
  goto err;
 }
 if ((ret = dbenv->set_data_dir(dbenv, data_dir)) != 0) {
  dbenv->err(dbenv, ret, "set_data_dir: %s", data_dir);
  goto err;
 }

 /* Open the environment with full transactional support. */
 if ((ret = dbenv->open(dbenv, home, DB_CREATE |
     DB_INIT_LOG | DB_INIT_LOCK | DB_INIT_MPOOL | DB_INIT_TXN, 0)) != 0) {
  dbenv->err(dbenv, ret, "environment open: %s", home);
  goto err;
 }

 return (dbenv);

err: (void)dbenv->close(dbenv, 0);
 return (NULL);
}
```

Opening databases within the environment

Once the environment has been created, database handles may be created and then opened within the environment. This is done by calling the db_create() function and specifying the appropriate environment as an argument.

File naming, database operations, and error handling will all be done as specified for the environment. For example, if the DB_INIT_LOCK or DB_INIT_CDB flags were specified when the environment was created or joined, database operations will automatically perform all necessary locking operations for the application.

The following is a simple example of opening two databases within a database environment:

```
DB_ENV *dbenv;
DB *dbp1, *dbp2;
int ret;

dbenv = NULL;
dbp1 = dbp2 = NULL;

/*
 * Create an environment and initialize it for additional error
 * reporting.
 */
if ((ret = db_env_create(&dbenv, 0)) != 0) {
 fprintf(errfp, "%s: %s\n", progname, db_strerror(ret));
 return (ret);
}

dbenv->set_errfile(dbenv, errfp);
dbenv->set_errpfx(dbenv, progname);

/* Open an environment with just a memory pool. */
if ((ret =
    dbenv->open(dbenv, home, DB_CREATE | DB_INIT_MPOOL, 0)) != 0) {
 dbenv->err(dbenv, ret, "environment open: %s", home);
 goto err;
}

/* Open database #1. */
if ((ret = db_create(&dbp1, dbenv, 0)) != 0) {
 dbenv->err(dbenv, ret, "database create");
 goto err;
}
if ((ret = dbp1->open(dbp1,
    NULL, DATABASE1, NULL, DB_BTREE, DB_CREATE, 0664)) != 0) {
 dbenv->err(dbenv, ret, "DB->open: %s", DATABASE1);
 goto err;
}

/* Open database #2. */
if ((ret = db_create(&dbp2, dbenv, 0)) != 0) {
 dbenv->err(dbenv, ret, "database create");
 goto err;
}
```

```
 if ((ret = dbp2->open(dbp2,
     NULL, DATABASE2, NULL, DB_HASH, DB_CREATE, 0664)) != 0) {
  dbenv->err(dbenv, ret, "DB->open: %s", DATABASE2);
  goto err;
 }

 return (0);

err: if (dbp2 != NULL)
  (void)dbp2->close(dbp2, 0);
 if (dbp1 != NULL)
  (void)dbp2->close(dbp1, 0);
 (void)dbenv->close(dbenv, 0);
 return (1);
}
```

Error support

Berkeley DB offers programmatic support for displaying error return values. The db_strerror() function returns a pointer to the error message corresponding to any Berkeley DB error return. This is similar to the ANSI C strerror interface, but can handle both system error returns and Berkeley DB-specific return values.

For example:

```
int ret;
if ((ret = dbenv->set_cachesize(dbenv, 0, 32 * 1024, 1)) != 0) {
 fprintf(stderr, "set_cachesize failed: %s\n", db_strerror(ret));
 return (1);
}
```

There are also two additional error methods: DB_ENV->err() and DB_ENV->errx(). These methods work like the ANSI C printf function, taking a printf-style format string and argument list, and writing a message constructed from the format string and arguments.

The DB_ENV->err() function appends the standard error string to the constructed message; the DB_ENV->errx() function does not.

Error messages can be configured always to include a prefix (for example, the program name) using the DB_ENV->set_errpfx() method.

These functions provide simpler ways of displaying Berkeley DB error messages:

```
int ret;
dbenv->set_errpfx(dbenv, program_name);
if ((ret = dbenv->open(dbenv, home,
    DB_CREATE | DB_INIT_LOG | DB_INIT_TXN | DB_USE_ENVIRON, 0))
    != 0) {
 dbenv->err(dbenv, ret, "open: %s", home);
 dbenv->errx(dbenv,
```

```
        "contact your system administrator: session ID was %d",
        session_id);
    return (1);
}
```

For example, if the program was called "my_app", and it tried to open an environment home directory in "/tmp/home" and the open call returned a permission error, the error messages shown would look like this:

```
my_app: open: /tmp/home: Permission denied.
my_app: contact your system administrator: session ID was 2
```

DB_CONFIG configuration file

Almost all of the configuration information that can be specified to DB_ENV class methods can also be specified using a configuration file. If a file named DB_CONFIG exists in the database home directory, it will be read for lines of the format **NAME VALUE**.

One or more whitespace characters are used to delimit the two parts of the line, and trailing whitespace characters are discarded. All empty lines or lines whose first character is a whitespace or hash (**#**) character will be ignored. Each line must specify both the NAME and the VALUE of the pair. The specific NAME VALUE pairs are documented in the manual for the corresponding methods (for example, the DB_ENV->set_data_dir() documentation includes NAME VALUE pair information Berkeley DB administrators can use to configure locations for database files).

The DB_CONFIG configuration file is intended to allow database environment administrators to customize environments independent of applications using the environment. For example, a database administrator can move the database log and data files to a different location without application recompilation. In addition, because the DB_CONFIG file is read when the database environment is opened, it can be used to overrule application configuration done before that time. For example a database administrator could override the compiled-in application cache size to a size more appropriate for a specific machine.

File naming

One of the most important tasks of the database environment is to structure file naming within Berkeley DB. Cooperating applications (or multiple invocations of the same application) must agree on the location of the database environment, log files and other files used by the Berkeley DB subsystems, and, of course, the database files. Although it is possible to specify full pathnames to all Berkeley DB methods, this is cumbersome and requires applications be recompiled when database files are moved.

Applications are normally expected to specify a single directory home for the database environment. This can be done easily in the call to DB_ENV->open() by specifying a value for the **db_home** argument. There are more complex configurations in which it may be desirable to override **db_home** or provide supplementary path information.

Specifying file naming to Berkeley DB

The following list describes the possible ways in which file naming information may be specified to the Berkeley DB library. The specific circumstances and order in which these ways are applied are described in a subsequent paragraph.

db_home

If the **db_home** argument to DB_ENV->open() is non-NULL, its value may be used as the database home, and files named relative to its path.

DB_HOME

If the DB_HOME environment variable is set when DB_ENV->open() is called, its value may be used as the database home, and files named relative to its path.

The DB_HOME environment variable is intended to permit users and system administrators to override application and installation defaults. For example::

```
env DB_HOME=/database/my_home application
```

Application writers are encouraged to support the **-h** option found in the supporting Berkeley DB utilities to let users specify a database home.

DB_ENV methods

There are three DB_ENV methods that affect file naming. The DB_ENV->set_data_dir() method specifies a directory to search for database files. The DB_ENV->set_lg_dir() method specifies a directory in which to create logging files. The DB_ENV->set_tmp_dir() method specifies a directory in which to create backing temporary files. These methods are intended to permit applications to customize a file location for a database. For example, an application writer can place data files and log files in different directories or instantiate a new log directory each time the application runs.

DB_CONFIG

The same information specified to the DB_ENV methods may also be specified using the DB_CONFIG configuration file.

Filename resolution in Berkeley DB

The following list describes the specific circumstances and order in which the different ways of specifying file naming information are applied. Berkeley DB filename processing proceeds sequentially through the following steps:

absolute pathnames

If the filename specified to a Berkeley DB function is an *absolute pathname*, that filename is used without modification by Berkeley DB.

On UNIX systems, an absolute pathname is defined as any pathname that begins with a leading slash (**/**).

On Windows systems, an absolute pathname is any pathname that begins with a leading slash or leading backslash (\); or any pathname beginning with a single alphabetic character, a colon and a leading slash or backslash (for example, `C:/tmp`).

DB_ENV methods, DB_CONFIG

If a relevant configuration string (for example, set_data_dir), is specified either by calling a DB_ENV method or as a line in the DB_CONFIG configuration file, the value is prepended to the filename. If the resulting filename is an absolute pathname, the filename is used without further modification by Berkeley DB.

db_home

If the application specified a non-NULL **db_home** argument to DB_ENV->open(), its value is prepended to the filename. If the resulting filename is an absolute pathname, the filename is used without further modification by Berkeley DB.

DB_HOME

If the **db_home** argument is NULL, the DB_HOME environment variable was set, and the application has set the appropriate DB_USE_ENVIRON or DB_USE_ENVIRON_ROOT flags, its value is prepended to the filename. If the resulting filename is an absolute pathname, the filename is used without further modification by Berkeley DB.

default

Finally, all filenames are interpreted relative to the current working directory of the process.

The common model for a Berkeley DB environment is one in which only the DB_HOME environment variable, or the **db_home** argument is specified. In this case, all data filenames are relative to that directory, and all files created by the Berkeley DB subsystems will be created in that directory.

The more complex model for a transaction environment might be one in which a database home is specified, using either the DB_HOME environment variable or the **db_home** argument to DB_ENV->open(); and then the data directory and logging directory are set to the relative pathnames of directories underneath the environment home.

Examples

Store all files in the directory `/a/database`:

```
dbenv->open(dbenv, "/a/database", flags, mode);
```

Create temporary backing files in `/b/temporary`, and all other files in `/a/database`:

```
dbenv->set_tmp_dir(dbenv, "/b/temporary");
dbenv->open(dbenv, "/a/database", flags, mode);
```

Store data files in `/a/database/datadir`, log files in `/a/database/logdir`, and all other files in the directory `/a/database`:

```
dbenv->set_lg_dir(dbenv, "logdir");
dbenv->set_data_dir(dbenv, "datadir");
dbenv->open(dbenv, "/a/database", flags, mode);
```

Store data files in `/a/database/data1` and `/b/data2`, and all other files in the directory `/a/database`. Any data files that are created will be created in `/b/data2`, because it is the first data file directory specified:

```
dbenv->set_data_dir(dbenv, "/b/data2");
dbenv->set_data_dir(dbenv, "data1");
dbenv->open(dbenv, "/a/database", flags, mode);
```

Shared memory regions

Each of the Berkeley DB subsystems within an environment is described by one or more regions, or chunks of memory. The regions contain all of the per-process and per-thread shared information (including mutexes), that comprise a Berkeley DB environment. These regions are created in one of three types of memory, depending on the flags specified to the DB_ENV->open() method:

The system memory used by Berkeley DB is potentially useful past the lifetime of any particular process. Therefore, additional cleanup may be necessary after an application fails because there may be no way for Berkeley DB to ensure that system resources backing the shared memory regions are returned to the system.

The system memory that is used is architecture-dependent. For example, on systems supporting X/Open-style shared memory interfaces, such as UNIX systems, the `shmget`(2) and related System V IPC interfaces are used. Additionally, VxWorks systems use system memory. In these cases, an initial segment ID must be specified by the application to ensure that applications do not overwrite each other's database environments, so that the number of segments created does not grow without bounds. See the DB_ENV->set_shm_key() method for more information.

On Windows platforms, the use of the DB_SYSTEM_MEM flag is problematic because the operating system uses reference counting to clean up shared objects in the paging file automatically. In addition, the default access permissions for shared objects are different from files, which may cause problems when an environment is accessed by multiple processes running as different users. See Windows notes (page 319) for more information.

1. If the DB_PRIVATE flag is specified to the DB_ENV->open() method, regions are created in per-process heap memory; that is, memory returned by `malloc`(3).

   This flag should not be specified if more than a single process is accessing the environment because it is likely to cause database corruption and unpredictable behavior. For example, if both a server application and Berkeley DB utilities (for example, the db_archive utility, the db_checkpoint utility or the db_stat utility) are expected to access the environment, the DB_PRIVATE flag should not be specified.

2. If the DB_SYSTEM_MEM flag is specified to DB->open(), shared regions are created in system memory rather than files. This is an alternative mechanism for sharing the Berkeley DB environment among multiple processes and multiple threads within processes.

3. If no memory-related flags are specified to DB_ENV->open(), memory backed by the filesystem is used to store the regions. On UNIX systems, the Berkeley DB library will use the POSIX mmap interface. If mmap is not available, the UNIX shmget interfaces may be used instead, if they are available.

Any files created in the filesystem to back the regions are created in the environment home directory specified to the DB_ENV->open() call. These files are named __db.### (for example,

__db.001, __db.002 and so on). When region files are backed by the filesystem, one file per region is created. When region files are backed by system memory, a single file will still be created because there must be a well-known name in the filesystem so that multiple processes can locate the system shared memory that is being used by the environment.

Statistics about the shared memory regions in the environment can be displayed using the **-e** option to the db_stat utility.

Security

The following are security issues that should be considered when writing Berkeley DB applications:

Database environment permissions

The directory used as the Berkeley DB database environment should have its permissions set to ensure that files in the environment are not accessible to users without appropriate permissions. Applications that add to the user's permissions (for example, UNIX setuid or setgid applications), must be carefully checked to not permit illegal use of those permissions such as general file access in the environment directory.

Environment variables

Setting the DB_USE_ENVIRON and DB_USE_ENVIRON_ROOT flags and allowing the use of environment variables during file naming can be dangerous. Setting those flags in Berkeley DB applications with additional permissions (for example, UNIX setuid or setgid applications) could potentially allow users to read and write databases to which they would not normally have access.

File permissions

By default, Berkeley DB always creates files readable and writable by the owner and the group (that is, S_IRUSR, S_IWUSR, S_IRGRP and S_IWGRP; or octal mode 0660 on historic UNIX systems). The group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB.

Temporary backing files

If an unnamed database is created and the cache is too small to hold the database in memory, Berkeley DB will create a temporary physical file to enable it to page the database to disk as needed. In this case, environment variables such as **TMPDIR** may be used to specify the location of that temporary file. Although temporary backing files are created readable and writable by the owner only (S_IRUSR and S_IWUSR, or octal mode 0600 on historic UNIX systems), some filesystems may not sufficiently protect temporary files created in random directories from improper access. To be absolutely safe, applications storing sensitive data in unnamed databases should use the DB_ENV->set_tmp_dir() method to specify a temporary directory with known permissions.

Tcl API

The Berkeley DB Tcl API does not attempt to avoid evaluating input as Tcl commands. For this reason, it may be dangerous to pass unreviewed user input through the Berkeley DB Tcl API, as the input may subsequently be evaluated as a Tcl command. Additionally, the Berkeley DB Tcl API initialization routine resets process' effective user and group

IDs to the real user and group IDs, to minimize the effectiveness of a Tcl injection attack.

Encryption

Berkeley DB optionally supports encryption using the Rijndael/AES (also known as the Advanced Encryption Standard and Federal Information Processing Standard (FIPS) 197) algorithm for encryption or decryption. The algorithm is configured to use a 128-bit key. Berkeley DB uses a 16-byte initialization vector generated using the Mersenne Twister. All encrypted information is additionally checksummed using the SHA1 Secure Hash Algorithm, using a 160-bit message digest.

The encryption support provided with Berkeley DB is intended to protect applications from an attacker obtaining physical access to the media on which a Berkeley DB database is stored, or an attacker compromising a system on which Berkeley DB is running but who is unable to read system or process memory on that system. **The encryption support provided with Berkeley DB will not protect applications from attackers able to read system memory on the system where Berkeley DB is running.**

Encryption is not the default for created databases, even in database environments configured for encryption. In addition to configuring for encryption by calling the DB_ENV->set_encrypt() or DB->set_encrypt() methods, applications must specify the DB_ENCRYPT flag before creating the database in order for the database to be encrypted. Further, databases cannot be converted to an encrypted format after they have been created without dumping and re-creating them. Finally, encrypted databases cannot be read on systems with a different endianness than the system that created the encrypted database.

Each encrypted database environment (including all its encrypted databases) is encrypted using a single password and a single algorithm. Applications wanting to provide a finer granularity of database access must either use multiple database environments or implement additional access controls outside of Berkeley DB.

The only encrypted parts of a database environment are its databases and its log files. Specifically, the Shared memory regions (page 131) supporting the database environment are not encrypted. For this reason, it may be possible for an attacker to read some or all of an encrypted database by reading the on-disk files that back these shared memory regions. To prevent such attacks, applications may want to use in-memory filesystem support (on systems that support it), or the DB_PRIVATE or DB_SYSTEM_MEM flags to the DB_ENV->open() method, to place the shared memory regions in memory that is never written to a disk. As some systems page system memory to a backing disk, it is important to consider the specific operating system running on the machine as well. Finally, when backing database environment shared regions with the filesystem, Berkeley DB can be configured to overwrite the shared regions before removing them by specifying the DB_OVERWRITE flag. This option is only effective in the presence of fixed-block filesystems, journaling or logging filesystems will require operating system support and probably modification of the Berkeley DB sources.

While all user data is encrypted, parts of the databases and log files in an encrypted environment are maintained in an unencrypted state. Specifically, log record headers are not encrypted, only the actual log records. Additionally, database internal page header fields are not encrypted.

These page header fields includes information such as the page's DB_LSN number and position in the database's sort order.

Log records distributed by a replication master to replicated clients are transmitted to the clients in unencrypted form. If encryption is desired in a replicated application, the use of a secure transport is strongly suggested.

We gratefully acknowledge:

- Vincent Rijmen, Antoon Bosselaers and Paulo Barreto for writing the Rijndael/AES code used in Berkeley DB.
- Steve Reid and James H. Brown for writing the SHA1 checksum code used in Berkeley DB.
- Makoto Matsumoto and Takuji Nishimura for writing the Mersenne Twister code used in Berkeley DB.
- Adam Stubblefield for integrating the Rijndael/AES, SHA1 checksum and Mersenne Twister code into Berkeley DB.

Remote filesystems

When Berkeley DB database environment shared memory regions are backed by the filesystem, it is a common application error to create database environments backed by remote filesystems such as the Network File System (NFS), Windows network shares (SMB/CIFS) or the Andrew File System (AFS). Remote filesystems rarely support mapping files into process memory, and even more rarely support correct semantics for mutexes if the mapping succeeds. For this reason, we recommend database environment directories be created in a local filesystem.

For remote filesystems that do allow remote files to be mapped into process memory, database environment directories accessed via remote filesystems cannot be used simultaneously from multiple clients (that is, from multiple computers). No commercial remote filesystem of which we're aware supports coherent, distributed shared memory for remote-mounted files. As a result, different machines will see different versions of these shared region files, and the behavior is undefined.

Databases, log files, and temporary files may be placed on remote filesystems, as long as the remote filesystem fully supports standard POSIX filesystem semantics (although the application may incur a performance penalty for doing so). Further, read-only databases on remote filesystems can be accessed from multiple systems simultaneously. However, it is difficult (or impossible) for modifiable databases on remote filesystems to be accessed from multiple systems simultaneously. The reason is the Berkeley DB library caches modified database pages, and when those modified pages are written to the backing file is not entirely under application control. If two systems were to write database pages to the remote filesystem at the same time, database corruption could result. If a system were to write a database page back to the remote filesystem at the same time as another system read a page, a core dump in the reader could result.

FreeBSD note:

Some historic FreeBSD releases will return ENOLCK from fsync and close calls on NFS-mounted filesystems, even though the call has succeeded. To support Berkeley DB on these releases, the Berkeley DB code should be modified to ignore ENOLCK errors, or no Berkeley DB files should be placed on NFS-mounted filesystems on these systems. Note that current FreeBSD releases do not suffer from this problem.

Linux note:

Some historic Linux releases do not support complete semantics for the POSIX fsync call on NFS-mounted filesystems. No Berkeley DB files should be placed on NFS-mounted filesystems on these systems. Note that current Linux releases do not suffer from this problem.

Environment FAQ

1. **I'm using multiple processes to access an Berkeley DB database environment; is there any way to ensure that two processes don't run transactional recovery at the same time, or that all processes have exited the database environment so that recovery can be run?**

   See Handling failure in Transactional Data Store applications (page 145) and Architecting Transactional Data Store applications (page 146) for a full discussion of this topic.

2. **How can I associate application information with a DB or DB_ENV handle?**

   In the C API, the DB and DB_ENV structures each contain an "app_private" field intended to be used to reference application-specific information. See the db_create() and db_env_create() documentation for more information.

   In the C++ or Java APIs, the easiest way to associate application-specific data with a handle is to subclass the Db or DbEnv, for example subclassing Db to get MyDb. Objects of type MyDb will still have the Berkeley DB API methods available on them, and you can put any extra data or methods you want into the MyDb class. If you are using "callback" APIs that take Db or DbEnv arguments (for example, Db::set_bt_compare()) these will always be called with the Db or DbEnv objects you create. So if you always use MyDb objects, you will be able to take the first argument to the callback function and cast it to a MyDb (in C++, cast it to (MyDb*)). That will allow you to access your data members or methods.

Chapter 10. Berkeley DB Concurrent Data Store Applications

Concurrent Data Store introduction

It is often desirable to have concurrent read-write access to a database when there is no need for full recoverability or transaction semantics. For this class of applications, Berkeley DB provides interfaces supporting deadlock-free, multiple-reader/single writer access to the database. This means that at any instant in time, there may be either multiple readers accessing data or a single writer modifying data. The application is entirely unaware of which is happening, and Berkeley DB implements the necessary locking and blocking to ensure this behavior.

To create Berkeley DB Concurrent Data Store applications, you must first initialize an environment by calling DB_ENV->open(). You must specify the DB_INIT_CDB and DB_INIT_MPOOL flags flags to that method. It is an error to specify any of the other DB_ENV->open() subsystem or recovery configuration flags, for example, DB_INIT_LOCK, DB_INIT_TXN or DB_RECOVER All databases must, of course, be created in this environment by using the db_create() function or Db constructor, and specifying the environment as an argument.

Berkeley DB performs appropriate locking so that safe enforcement of the deadlock-free, multiple-reader/single-writer semantic is transparent to the application. However, a basic understanding of Berkeley DB Concurrent Data Store locking behavior is helpful when writing Berkeley DB Concurrent Data Store applications.

Berkeley DB Concurrent Data Store avoids deadlocks without the need for a deadlock detector by performing all locking on an entire database at once (or on an entire environment in the case of the DB_CDB_ALLDB flag), and by ensuring that at any given time only one thread of control is allowed to simultaneously hold a read (shared) lock and attempt to acquire a write (exclusive) lock.

All open Berkeley DB cursors hold a read lock, which serves as a guarantee that the database will not change beneath them; likewise, all non-cursor DB->get() operations temporarily acquire and release a read lock that is held during the actual traversal of the database. Because read locks will not conflict with each other, any number of cursors in any number of threads of control may be open simultaneously, and any number of DB->get() operations may be concurrently in progress.

To enforce the rule that only one thread of control at a time can attempt to upgrade a read lock to a write lock, however, Berkeley DB must forbid multiple cursors from attempting to write concurrently. This is done using the DB_WRITECURSOR flag to the DB->cursor() method. This is the only difference between access method calls in Berkeley DB Concurrent Data Store and in the other Berkeley DB products. The DB_WRITECURSOR flag causes the newly created cursor to be a "write" cursor; that is, a cursor capable of performing writes as well as reads. Only cursors thus created are permitted to perform write operations (either deletes or puts), and only one such cursor can exist at any given time.

Any attempt to create a second write cursor or to perform a non-cursor write operation while a write cursor is open will block until that write cursor is closed. Read cursors may open and

perform reads without blocking while a write cursor is extant. However, any attempts to actually perform a write, either using the write cursor or directly using the DB->put() or DB->del() methods, will block until all read cursors are closed. This is how the multiple-reader/single-writer semantic is enforced, and prevents reads from seeing an inconsistent database state that may be an intermediate stage of a write operation.

By default, Berkeley DB Concurrent Data Store does locking on a per-database basis. For this reason, using cursors to access multiple databases in different orders in different threads or processes, or leaving cursors open on one database while accessing another database, can cause an application to hang. If this behavior is a requirement for the application, Berkeley DB should be configured to do locking on an environment-wide basis. See the DB_CDB_ALLDB flag of the DB_ENV->set_flags() method for more information.

With these behaviors, Berkeley DB can guarantee deadlock-free concurrent database access, so that multiple threads of control are free to perform reads and writes without needing to handle synchronization themselves or having to run a deadlock detector. Berkeley DB has no direct knowledge of which cursors belong to which threads, so some care must be taken to ensure that applications do not inadvertently block themselves, causing the application to hang and be unable to proceed.

As a consequence of the Berkeley DB Concurrent Data Store locking model, the following sequences of operations will cause a thread to block itself indefinitely:

1. Keeping a cursor open while issuing a DB->put() or DB->del() access method call.

2. Attempting to open a write cursor while another cursor is already being held open by the same thread of control. Note that it is correct operation for one thread of control to attempt to open a write cursor or to perform a non-cursor write (DB->put() or DB->del()) while a write cursor is already active in another thread. It is only a problem if these things are done within a single thread of control -- in which case that thread will block and never be able to release the lock that is blocking it.

3. Not testing Berkeley DB error return codes (if any cursor operation returns an unexpected error, that cursor must still be closed).

If the application needs to open multiple cursors in a single thread to perform an operation, it can indicate to Berkeley DB that the cursor locks should not block each other by creating a Berkeley DB Concurrent Data Store **group**, using DB_ENV->cdsgroup_begin(). This creates a locker ID that is shared by all cursors opened in the group.

Berkeley DB Concurrent Data Store groups use a TXN handle to indicate the shared locker ID to Berkeley DB calls, and call DB_TXN->commit() to end the group. This is a convenient way to pass the locked ID to the calls where it is needed, but should not be confused with the real transactional semantics provided by Berkeley DB Transactional Data Store. In particular, Berkeley DB Concurrent Data Store groups do not provide any abort or recovery facilities, and have no impact on durability of operations.

Handling failure in Data Store and Concurrent Data Store applications

When building Data Store and Concurrent Data Store applications, there are design issues to consider whenever a thread of control with open Berkeley DB handles fails for any reason (where a thread of control may be either a true thread or a process).

The simplest case is handling system failure for any Data Store or Concurrent Data Store application. In the case of system failure, it doesn't matter if the application has opened a database environment or is just using standalone databases: if the system fails, after the application has modified a database and has not subsequently flushed the database to stable storage (by calling either the DB->close(), DB->sync() or DB_ENV->memp_sync() methods), the database may be left in a corrupted state. In this case, before accessing the database again, the database should either be:

- removed and re-created,
- removed and restored from the last known good backup, or
- verified using the DB->verify() method or db_verify utility utility. If the database does not verify cleanly, the contents may be salvaged using the **-R** and **-r** options of the db_dump utility.

Applications where the potential for data loss is unacceptable should consider the Berkeley DB Transactional Data Store product, which offers standard transactional durability guarantees, including recoverability after failure.

Additionally, system failure requires that any persistent database environment (that is, any database environment not created using the DB_PRIVATE flag), be removed. Database environments may be removed using the DB_ENV->remove() method. If the persistent database environment was backed by the filesystem (that is, the environment was not created using the DB_SYSTEM_MEM flag), the database environment may also be safely removed by deleting the environment's files with standard system utilities.

The second case is application failure for a Data Store application, with or without a database environment, or application failure for a Concurrent Data Store application without a database environment: as in the case of system failure, if any thread of control fails, after the application has modified a database and has not subsequently flushed the database to stable storage, the database may be left in a corrupted state. In this case, the database should be handled as described previously in the system failure case.

The third case is application failure for a Concurrent Data Store application with a database environment. There are resources maintained in database environments that may be left locked if a thread of control exits without first closing all open Berkeley DB handles. Concurrent Data Store applications with database environments have an additional option for handling the unexpected exit of a thread of control, the DB_ENV->failchk() method.

The DB_ENV->failchk() will return DB_RUNRECOVERY (page 230) if the database environment is unusable as a result of the thread of control failure. (If a data structure mutex or a database

write lock is left held by thread of control failure, the application should not continue to use the database environment, as subsequent use of the environment is likely to result in threads of control convoying behind the held locks.) The DB_ENV->failchk() call will release any database read locks that have been left held by the exit of a thread of control. In this case, the application can continue to use the database environment.

A Concurrent Data Store application recovering from a thread of control failure should call DB_ENV->failchk(), and, if it returns success, the application can continue. If DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the application should proceed as described for the case of system failure.

Architecting Data Store and Concurrent Data Store applications

When building Data Store and Concurrent Data Store applications, the architecture decisions involve application startup (cleaning up any existing databases, the removal of any existing database environment and creation of a new environment), and handling system or application failure. "Cleaning up" databases involves removal and re-creation of the database, restoration from an archival copy and/or verification and optional salvage, as described in Handling failure in Data Store and Concurrent Data Store applications (page 138).

Data Store or Concurrent Data Store applications without database environments are single process, by definition. These applications should start up, re-create, restore, or verify and optionally salvage their databases and run until eventual exit or application or system failure. After system or application failure, that process can simply repeat this procedure. This document will not discuss the case of these applications further.

Otherwise, the first question of Data Store and Concurrent Data Store architecture is the cleaning up existing databases and the removal of existing database environments, and the subsequent creation of a new environment. For obvious reasons, the application must serialize the re-creation, restoration, or verification and optional salvage of its databases. Further, environment removal and creation must be single-threaded, that is, one thread of control (where a thread of control is either a true thread or a process) must remove and re-create the environment before any other thread of control can use the new environment. It may simplify matters that Berkeley DB serializes creation of the environment, so multiple threads of control attempting to create a environment will serialize behind a single creating thread.

Removing a database environment will first mark the environment as "failed", causing any threads of control still running in the environment to fail and return to the application. This feature allows applications to remove environments without concern for threads of control that might still be running in the removed environment.

One consideration in removing a database environment which may be in use by another thread, is the type of mutex being used by the Berkeley DB library. In the case of database environment failure when using test-and-set mutexes, threads of control waiting on a mutex when the environment is marked "failed" will quickly notice the failure and will return an error from the Berkeley DB API. In the case of environment failure when using blocking mutexes, where the underlying system mutex implementation does not unblock mutex waiters after the thread of control holding the mutex dies, threads waiting on a mutex when an environment is recovered might hang forever. Applications blocked on events (for example, an application blocked on a

network socket or a GUI event) may also fail to notice environment recovery within a reasonable amount of time. Systems with such mutex implementations are rare, but do exist; applications on such systems should use an application architecture where the thread recovering the database environment can explicitly terminate any process using the failed environment, or configure Berkeley DB for test-and-set mutexes, or incorporate some form of long-running timer or watchdog process to wake or kill blocked processes should they block for too long.

Regardless, it makes little sense for multiple threads of control to simultaneously attempt to remove and re-create a environment, since the last one to run will remove all environments created by the threads of control that ran before it. However, for some few applications, it may make sense for applications to have a single thread of control that checks the existing databases and removes the environment, after which the application launches a number of processes, any of which are able to create the environment.

With respect to cleaning up existing databases, the database environment must be removed before the databases are cleaned up. Removing the environment causes any Berkeley DB library calls made by threads of control running in the failed environment to return failure to the application. Removing the database environment first ensures the threads of control in the old environment do not race with the threads of control cleaning up the databases, possibly overwriting them after the cleanup has finished. Where the application architecture and system permit, many applications kill all threads of control running in the failed database environment before removing the failed database environment, on general principles as well as to minimize overall system resource usage. It does not matter if the new environment is created before or after the databases are cleaned up.

After having dealt with database and database environment recovery after failure, the next issue to manage is application failure. As described in Handling failure in Data Store and Concurrent Data Store applications (page 138), when a thread of control in a Data Store or Concurrent Data Store application fails, it may exit holding data structure mutexes or logical database locks. These mutexes and locks must be released to avoid the remaining threads of control hanging behind the failed thread of control's mutexes or locks.

There are three common ways to architect Berkeley DB Data Store and Concurrent Data Store applications. The one chosen is usually based on whether or not the application is comprised of a single process or group of processes descended from a single process (for example, a server started when the system first boots), or if the application is comprised of unrelated processes (for example, processes started by web connections or users logging into the system).

1. The first way to architect Data Store and Concurrent Data Store applications is as a single process (the process may or may not be multithreaded.)
   When this process starts, it removes any existing database environment and creates a new environment. It then cleans up the databases and opens those databases in the environment. The application can subsequently create new threads of control as it chooses. Those threads of control can either share already open Berkeley DB DB_ENV and DB handles, or create their own. In this architecture, databases are rarely opened or closed when more than a single thread of control is running; that is, they are opened when only a single thread is running, and closed after all threads but one have exited. The last thread of control to exit closes the databases and the database environment.

This architecture is simplest to implement because thread serialization is easy and failure detection does not require monitoring multiple processes.

If the application's thread model allows the process to continue after thread failure, the DB_ENV->failchk() method can be used to determine if the database environment is usable after the failure. If the application does not call DB_ENV->failchk(), or DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the application must behave as if there has been a system failure, removing the environment and creating a new environment, and cleaning up any databases it wants to continue to use. Once these actions have been taken, other threads of control can continue (as long as all existing Berkeley DB handles are first discarded), or restarted.

2. The second way to architect Data Store and Concurrent Data Store applications is as a group of related processes (the processes may or may not be multithreaded).
   This architecture requires the order in which threads of control are created be controlled to serialize database environment removal and creation, and database cleanup.

   In addition, this architecture requires that threads of control be monitored. If any thread of control exits with open Berkeley DB handles, the application may call the DB_ENV->failchk() method to determine if the database environment is usable after the exit. If the application does not call DB_ENV->failchk(), or DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the application must behave as if there has been a system failure, removing the environment and creating a new environment, and cleaning up any databases it wants to continue to use. Once these actions have been taken, other threads of control can continue (as long as all existing Berkeley DB handles are first discarded), or restarted.

   The easiest way to structure groups of related processes is to first create a single "watcher" process (often a script) that starts when the system first boots, removes and creates the database environment, cleans up the databases and then creates the processes or threads that will actually perform work. The initial thread has no further responsibilities other than to wait on the threads of control it has started, to ensure none of them unexpectedly exit. If a thread of control exits, the watcher process optionally calls the DB_ENV->failchk() method. If the application does not call DB_ENV->failchk(), or if DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the environment can no longer be used, the watcher kills all of the threads of control using the failed environment, cleans up, and starts new threads of control to perform work.

3. The third way to architect Data Store and Concurrent Data Store applications is as a group of unrelated processes (the processes may or may not be multithreaded). This is the most difficult architecture to implement because of the level of difficulty in some systems of finding and monitoring unrelated processes.
   One solution is to log a thread of control ID when a new Berkeley DB handle is opened. For example, an initial "watcher" process could open/create the database environment, clean up the databases and then create a sentinel file. Any "worker" process wanting to use the environment would check for the sentinel file. If the sentinel file does not exist, the worker would fail or wait for the sentinel file to be created. Once the sentinel file exists, the worker would register its process ID with the watcher (via shared memory, IPC or some other registry mechanism), and then the worker would open its DB_ENV handles and proceed. When the worker finishes using the environment, it would unregister its process ID with the watcher. The watcher periodically checks to ensure that no worker has failed while using the

environment. If a worker fails while using the environment, the watcher removes the sentinel file, kills all of the workers currently using the environment, cleans up the environment and databases, and finally creates a new sentinel file.

The weakness of this approach is that, on some systems, it is difficult to determine if an unrelated process is still running. For example, POSIX systems generally disallow sending signals to unrelated processes. The trick to monitoring unrelated processes is to find a system resource held by the process that will be modified if the process dies. On POSIX systems, flock- or fcntl-style locking will work, as will LockFile on Windows systems. Other systems may have to use other process-related information such as file reference counts or modification times. In the worst case, threads of control can be required to periodically re-register with the watcher process: if the watcher has not heard from a thread of control in a specified period of time, the watcher will take action, cleaning up the environment.

If it is not practical to monitor the processes sharing a database environment, it may be possible to monitor the environment to detect if a thread of control has failed holding open Berkeley DB handles. This would be done by having a "watcher" process periodically call the DB_ENV->failchk() method. If DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the watcher would then take action, cleaning up the environment.

The weakness of this approach is that all threads of control using the environment must specify an "ID" function and an "is-alive" function using the DB_ENV->set_thread_id() method. (In other words, the Berkeley DB library must be able to assign a unique ID to each thread of control, and additionally determine if the thread of control is still running. It can be difficult to portably provide that information in applications using a variety of different programming languages and running on a variety of different platforms.)

Obviously, when implementing a process to monitor other threads of control, it is important the watcher process' code be as simple and well-tested as possible, because the application may hang if it fails.

Chapter 11. Berkeley DB Transactional Data Store Applications

Transactional Data Store introduction

It is difficult to write a useful transactional tutorial and still keep within reasonable bounds of documentation; that is, without writing a book on transactional programming. We have two goals in this section: to familiarize readers with the transactional interfaces of Berkeley DB and to provide code building blocks that will be useful for creating applications.

We have not attempted to present this information using a real-world application. First, transactional applications are often complex and time-consuming to explain. Also, one of our goals is to give you an understanding of the wide variety of tools Berkeley DB makes available to you, and no single application would use most of the interfaces included in the Berkeley DB library. For these reasons, we have chosen to simply present the Berkeley DB data structures and programming solutions, using examples that differ from page to page. All the examples are included in a standalone program you can examine, modify, and run; and from which you will be able to extract code blocks for your own applications. Fragments of the program will be presented throughout this chapter, and the complete text of the example program [transapp.cs] for IEEE/ANSI Std 1003.1 (POSIX) standard systems is included in the Berkeley DB distribution.

Why transactions?

Perhaps the first question to answer is "Why transactions?" There are a number of reasons to include transactional support in your applications. The most common ones are the following:

Recoverability

Applications often need to ensure that no matter how the system or application fails, previously saved data is available the next time the application runs. This is often called Durability.

Atomicity

Applications may need to make multiple changes to one or more databases, but ensure that either all of the changes happen, or none of them happens. Transactions guarantee that a group of changes are atomic; that is, if the application or system fails, either all of the changes to the databases will appear when the application next runs, or none of them.

Isolation

Applications may need to make changes in isolation, that is, ensure that only a single thread of control is modifying a key/data pair at a time. Transactions ensure each thread of control sees all records as if all other transactions either completed before or after its transaction.

Terminology

Here are some definitions that will be helpful in understanding transactions:

Thread of control

Berkeley DB is indifferent to the type or style of threads being used by the application; or, for that matter, if threads are being used at all — because Berkeley DB supports multiprocess access. In the Berkeley DB documentation, any time we refer to a *thread of control*, it can be read as a true thread (one of many in an application's address space) or a process.

Free-threaded

A Berkeley DB handle that can be used by multiple threads simultaneously without any application-level synchronization is called *free-threaded*.

Transaction

A *transaction* is a one or more operations on one or more databases that should be treated as a single unit of work. For example, changes to a set of databases, in which either all of the changes must be applied to the database(s) or none of them should. Applications specify when each transaction starts, what database operations are included in it, and when it ends.

Transaction abort/commit

Every transaction ends by *committing* or *aborting*. If a transaction commits, Berkeley DB guarantees that any database changes included in the transaction will never be lost, even after system or application failure. If a transaction aborts, or is uncommitted when the system or application fails, then the changes involved will never appear in the database.

System or application failure

System or application failure is the phrase we use to describe something bad happening near your data. It can be an application dumping core, being interrupted by a signal, the disk filling up, or the entire system crashing. In any case, for whatever reason, the application can no longer make forward progress, and its databases are left in an unknown state.

Recovery

Recovery is what makes the database consistent after a system or application failure. The recovery process includes review of log files and databases to ensure that the changes from each committed transaction appear in the database, and that no changes from an unfinished (or aborted) transaction do. Whenever system or application failure occurs, applications must usually run recovery.

Deadlock

Deadlock, in its simplest form, happens when one thread of control owns resource A, but needs resource B; while another thread of control owns resource B, but needs resource A. Neither thread of control can make progress, and so one has to give up and release all its resources, at which time the remaining thread of control can make forward progress.

Handling failure in Transactional Data Store applications

When building Transactional Data Store applications, there are design issues to consider whenever a thread of control with open Berkeley DB handles fails for any reason (where a thread of control may be either a true thread or a process).

The first case is handling system failure: if the system fails, the database environment and the databases may be left in a corrupted state. In this case, recovery must be performed on the database environment before any further action is taken, in order to:

- recover the database environment resources,
- release any locks or mutexes that may have been held to avoid starvation as the remaining threads of control convoy behind the held locks, and
- resolve any partially completed operations that may have left a database in an inconsistent or corrupted state.

For details on performing recovery, see the Recovery procedures (page 176).

The second case is handling the failure of a thread of control. There are resources maintained in database environments that may be left locked or corrupted if a thread of control exits unexpectedly. These resources include data structure mutexes, logical database locks and unresolved transactions (that is, transactions which were never aborted or committed). While Transactional Data Store applications can treat the failure of a thread of control in the same way as they do a system failure, they have an alternative choice, the DB_ENV->failchk() method.

The DB_ENV->failchk() will return DB_RUNRECOVERY (page 230) if the database environment is unusable as a result of the thread of control failure. (If a data structure mutex or a database write lock is left held by thread of control failure, the application should not continue to use the database environment, as subsequent use of the environment is likely to result in threads of control convoying behind the held locks.) The DB_ENV->failchk() call will release any database read locks that have been left held by the exit of a thread of control, and abort any unresolved transactions. In this case, the application can continue to use the database environment.

A Transactional Data Store application recovering from a thread of control failure should call DB_ENV->failchk(), and, if it returns success, the application can continue. If DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the application should proceed as described for the case of system failure.

It greatly simplifies matters that recovery may be performed regardless of whether recovery needs to be performed; that is, it is not an error to recover a database environment for which recovery is not strictly necessary. For this reason, applications should not try to determine if the database environment was active when the application or system failed. Instead, applications should run recovery any time the DB_ENV->failchk() method returns DB_RUNRECOVERY (page 230), or, if the application is not calling the DB_ENV->failchk() method, any time any thread of control accessing the database environment fails, as well as any time the system reboots.

Architecting Transactional Data Store applications

When building Transactional Data Store applications, the architecture decisions involve application startup (running recovery) and handling system or application failure. For details on performing recovery, see the Recovery procedures (page 176).

Recovery in a database environment is a single-threaded procedure, that is, one thread of control or process must complete database environment recovery before any other thread of control or process operates in the Berkeley DB environment.

Performing recovery first marks any existing database environment as "failed" and then removes it, causing threads of control running in the database environment to fail and return to the application. This feature allows applications to recover environments without concern for threads of control that might still be running in the removed environment. The subsequent re-creation of the database environment is serialized, so multiple threads of control attempting to create a database environment will serialize behind a single creating thread.

One consideration in removing (as part of recovering) a database environment which may be in use by another thread, is the type of mutex being used by the Berkeley DB library. In the case of database environment failure when using test-and-set mutexes, threads of control waiting on a mutex when the environment is marked "failed" will quickly notice the failure and will return an error from the Berkeley DB API. In the case of environment failure when using blocking mutexes, where the underlying system mutex implementation does not unblock mutex waiters after the thread of control holding the mutex dies, threads waiting on a mutex when an environment is recovered might hang forever. Applications blocked on events (for example, an application blocked on a network socket, or a GUI event) may also fail to notice environment recovery within a reasonable amount of time. Systems with such mutex implementations are rare, but do exist; applications on such systems should use an application architecture where the thread recovering the database environment can explicitly terminate any process using the failed environment, or configure Berkeley DB for test-and-set mutexes, or incorporate some form of long-running timer or watchdog process to wake or kill blocked processes should they block for too long.

Regardless, it makes little sense for multiple threads of control to simultaneously attempt recovery of a database environment, since the last one to run will remove all database environments created by the threads of control that ran before it. However, for some applications, it may make sense for applications to have a single thread of control that performs recovery and then removes the database environment, after which the application launches a number of processes, any of which will create the database environment and continue forward.

There are three common ways to architect Berkeley DB Transactional Data Store applications. The one chosen is usually based on whether or not the application is comprised of a single process or group of processes descended from a single process (for example, a server started when the system first boots), or if the application is comprised of unrelated processes (for example, processes started by web connections or users logged into the system).

1. The first way to architect Transactional Data Store applications is as a single process (the process may or may not be multithreaded.)

When this process starts, it runs recovery on the database environment and then opens its databases. The application can subsequently create new threads as it chooses. Those threads can either share already open Berkeley DB DB_ENV and DB handles, or create their own. In this architecture, databases are rarely opened or closed when more than a single thread of control is running; that is, they are opened when only a single thread is running, and closed after all threads but one have exited. The last thread of control to exit closes the databases and the database environment.

This architecture is simplest to implement because thread serialization is easy and failure detection does not require monitoring multiple processes.

If the application's thread model allows processes to continue after thread failure, the DB_ENV->failchk() method can be used to determine if the database environment is usable after thread failure. If the application does not call DB_ENV->failchk(), or DB_ENV->failchk() returns DB_RUNRECOVERY (page 230), the application must behave as if there has been a system failure, performing recovery and re-creating the database environment. Once these actions have been taken, other threads of control can continue (as long as all existing Berkeley DB handles are first discarded).

2. The second way to architect Transactional Data Store applications is as a group of related processes (the processes may or may not be multithreaded).

   This architecture requires the order in which threads of control are created be controlled to serialize database environment recovery.

   In addition, this architecture requires that threads of control be monitored. If any thread of control exits with open Berkeley DB handles, the application may call the DB_ENV->failchk() method to detect lost mutexes and locks and determine if the application can continue. If the application does not call DB_ENV->failchk(), or DB_ENV->failchk() returns that the database environment can no longer be used, the application must behave as if there has been a system failure, performing recovery and creating a new database environment. Once these actions have been taken, other threads of control can be continued (as long as all existing Berkeley DB handles are first discarded), or

   The easiest way to structure groups of related processes is to first create a single "watcher" process (often a script) that starts when the system first boots, runs recovery on the database environment and then creates the processes or threads that will actually perform work. The initial thread has no further responsibilities other than to wait on the threads of control it has started, to ensure none of them unexpectedly exit. If a thread of control exits, the watcher process optionally calls the DB_ENV->failchk() method. If the application does not call DB_ENV->failchk() or if DB_ENV->failchk() returns that the environment can no longer be used, the watcher kills all of the threads of control using the failed environment, runs recovery, and starts new threads of control to perform work.

3. The third way to architect Transactional Data Store applications is as a group of unrelated processes (the processes may or may not be multithreaded). This is the most difficult architecture to implement because of the level of difficulty in some systems of finding and monitoring unrelated processes. There are several possible techniques to implement this architecture.

One solution is to log a thread of control ID when a new Berkeley DB handle is opened. For example, an initial "watcher" process could run recovery on the database environment and then create a sentinel file. Any "worker" process wanting to use the environment would check for the sentinel file. If the sentinel file does not exist, the worker would fail or wait for the sentinel file to be created. Once the sentinel file exists, the worker would register its process ID with the watcher (via shared memory, IPC or some other registry mechanism), and then the worker would open its DB_ENV handles and proceed. When the worker finishes using the environment, it would unregister its process ID with the watcher. The watcher periodically checks to ensure that no worker has failed while using the environment. If a worker fails while using the environment, the watcher removes the sentinel file, kills all of the workers currently using the environment, runs recovery on the environment, and finally creates a new sentinel file.

The weakness of this approach is that, on some systems, it is difficult to determine if an unrelated process is still running. For example, POSIX systems generally disallow sending signals to unrelated processes. The trick to monitoring unrelated processes is to find a system resource held by the process that will be modified if the process dies. On POSIX systems, flock- or fcntl-style locking will work, as will LockFile on Windows systems. Other systems may have to use other process-related information such as file reference counts or modification times. In the worst case, threads of control can be required to periodically re-register with the watcher process: if the watcher has not heard from a thread of control in a specified period of time, the watcher will take action, recovering the environment.

The Berkeley DB library includes one built-in implementation of this approach, the DB_ENV->open() method's DB_REGISTER flag:

If the DB_REGISTER flag is set, each process opening the database environment first checks to see if recovery needs to be performed. If recovery needs to be performed for any reason (including the initial creation of the database environment), and DB_RECOVER is also specified, recovery will be performed and then the open will proceed normally. If recovery needs to be performed and DB_RECOVER is not specified, DB_RUNRECOVERY (page 230) will be returned. If recovery does not need to be performed, DB_RECOVER will be ignored.

Prior to the actual recovery beginning, the DB_EVENT_REG_PANIC event is set for the environment. Processes in the application using the DB_ENV->set_event_notify() method will be notified when they do their next operations in the environment. Processes receiving this event should exit the environment. Also, the DB_EVENT_REG_ALIVE event will be triggered if there are other processes currently attached to the environment. Only the process doing the recovery will receive this event notification. It will receive this notification once for each process still attached to the environment. The parameter of the DB_ENV->set_event_notify() callback will contain the process identifier of the process still attached. The process doing the recovery can then signal the attached process or perform some other operation prior to recovery (i.e. kill the attached process).

The DB_ENV->set_timeout() method's DB_SET_REG_TIMEOUT flag can be set to establish a wait period before starting recovery. This creates a window of time for other processes to receive the DB_EVENT_REG_PANIC event and exit the environment.

There are three additional requirements for the DB_REGISTER architecture to work:

- First, all applications using the database environment must specify the DB_REGISTER flag when opening the environment. However, there is no additional requirement if the application chooses a single process to recover the environment, as the first process to open the database environment will know to perform recovery.
- Second, there can only be a single DB_ENV handle per database environment in each process. As the DB_REGISTER locking is per-process, not per-thread, multiple DB_ENV handles in a single environment could race with each other, potentially causing data corruption.
- Third, the DB_REGISTER implementation does not explicitly terminate processes using the database environment which is being recovered. Instead, it relies on the processes themselves noticing the database environment has been discarded from underneath them. For this reason, the DB_REGISTER flag should be used with a mutex implementation that does not block in the operating system, as that risks a thread of control blocking forever on a mutex which will never be granted. Using any test-and-set mutex implementation ensures this cannot happen, and for that reason the DB_REGISTER flag is generally used with a test-and-set mutex implementation.

A second solution for groups of unrelated processes is also based on a "watcher process". This solution is intended for systems where it is not practical to monitor the processes sharing a database environment, but it is possible to monitor the environment to detect if a thread of control has failed holding open Berkeley DB handles. This would be done by having a "watcher" process periodically call the DB_ENV->failchk() method. If DB_ENV->failchk() returns that the environment can no longer be used, the watcher would then take action, recovering the environment.

The weakness of this approach is that all threads of control using the environment must specify an "ID" function and an "is-alive" function using the DB_ENV->set_thread_id() method. (In other words, the Berkeley DB library must be able to assign a unique ID to each thread of control, and additionally determine if the thread of control is still running. It can be difficult to portably provide that information in applications using a variety of different programming languages and running on a variety of different platforms.)

A third solution for groups of unrelated processes is a hybrid of the two above. Along with implementing the built-in sentinel approach with the the DB_ENV->open() methods DB_REGISTER flag, the DB_FAILCHK flag can be specified. When using both flags, each process opening the database environment first checks to see if recocvery needs to be performed. If recovery needs to be performed for any reason, it will first determine if a thread of control exited while holding database read locks, and release those. Then it will abort any unresolved transactions. If these steps are successful, the process opening the environment will continue without the need for any additional recocvery. If these steps are unsuccessful, then additional recovery will be performed if DB_RECOVER is specified and if DB_RECOVER is not specified, DB_RUNRECOVERY (page 230)will be returned.

Since this solution is hybrid of the first two, all of the requirements of both of them must be implemented (will need "ID" function, "is-alive" function, single DB_ENV handle per database, etc.)

The described approaches are different, and should not be combined. Applications might use either the DB_REGISTER approach, the DB_ENV->failchk() or the hybrid approach, but not together in the same application. For example, a POSIX application written as a library underneath a wide variety of interfaces and differing APIs might choose the DB_REGISTER approach for a few reasons: first, it does not require making periodic calls to the DB_ENV->failchk() method; second, when implementing in a variety of languages, is may be more difficult to specify unique IDs for each thread of control; third, it may be more difficult determine if a thread of control is still running, as any particular thread of control is likely to lack sufficient permissions to signal other processes. Alternatively, an application with a dedicated watcher process, running with appropriate permissions, might choose the DB_ENV->failchk() approach as supporting higher overall throughput and reliability, as that approach allows the application to abort unresolved transactions and continue forward without having to recover the database environment. The hybrid approach is useful in situations where running a dedicated watcher process is not practical but getting the equivalent of DB_ENV->failchk() on the DB_ENV->open() is important.

Obviously, when implementing a process to monitor other threads of control, it is important the watcher process' code be as simple and well-tested as possible, because the application may hang if it fails.

Opening the environment

Creating transaction-protected applications using the Berkeley DB library is quite easy. Applications first use DB_ENV->open() to initialize the database environment. Transaction-protected applications normally require all four Berkeley DB subsystems, so the DB_INIT_MPOOL, DB_INIT_LOCK, DB_INIT_LOG, and DB_INIT_TXN flags should be specified.

Once the application has called DB_ENV->open(), it opens its databases within the environment. Once the databases are opened, the application makes changes to the databases inside of transactions. Each set of changes that entails a unit of work should be surrounded by the appropriate DB_ENV->txn_begin(), DB_TXN->commit() and DB_TXN->abort() calls. The Berkeley DB access methods will make the appropriate calls into the Lock, Log and Memory Pool subsystems in order to guarantee transaction semantics. When the application is ready to exit, all outstanding transactions should have been committed or aborted.

Databases accessed by a transaction must not be closed during the transaction. Once all outstanding transactions are finished, all open Berkeley DB files should be closed. When the Berkeley DB database files have been closed, the environment should be closed by calling DB_ENV->close().

The following code fragment creates the database environment directory then opens the environment, running recovery. Our DB_ENV database environment handle is declared to be free-threaded using the DB_THREAD flag, and so may be used by any number of threads that we may subsequently create.

```
#include <sys/types.h>
#include <sys/stat.h>

#include <errno.h>
#include <stdarg.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <db.h>

#define ENV_DIRECTORY "TXNAPP"

void  env_dir_create(void);
void  env_open(DB_ENV **);

int
main(int argc, char *argv)
{
 extern int optind;
 DB *db_cats, *db_color, *db_fruit;
 DB_ENV *dbenv;
 int ch;

 while ((ch = getopt(argc, argv, "")) != EOF)
  switch (ch) {
  case '?':
  default:
   usage();
  }
 argc -= optind;
 argv += optind;

 env_dir_create();
 env_open(&dbenv);

 return (0);
}

void
env_dir_create()
{
 struct stat sb;

 /*
  * If the directory exists, we're done.  We do not further check
  * the type of the file, DB will fail appropriately if it's the
  * wrong type.
  */
 if (stat(ENV_DIRECTORY, &sb) == 0)
  return;

 /* Create the directory, read/write/access owner only. */
 if (mkdir(ENV_DIRECTORY, S_IRWXU) != 0) {
```

```
  fprintf(stderr,
      "txnapp: mkdir: %s: %s\n", ENV_DIRECTORY, strerror(errno));
  exit (1);
 }
}

void
env_open(DB_ENV **dbenvp)
{
 DB_ENV *dbenv;
 int ret;

 /* Create the environment handle. */
 if ((ret = db_env_create(&dbenv, 0)) != 0) {
  fprintf(stderr,
      "txnapp: db_env_create: %s\n", db_strerror(ret));
  exit (1);
 }

 /* Set up error handling. */
 dbenv->set_errpfx(dbenv, "txnapp");
 dbenv->set_errfile(dbenv, stderr);

 /*
  * Open a transactional environment:
  * create if it doesn't exist
  * free-threaded handle
  * run recovery
  * read/write owner only
  */
 if ((ret = dbenv->open(dbenv, ENV_DIRECTORY,
     DB_CREATE | DB_INIT_LOCK | DB_INIT_LOG |
     DB_INIT_MPOOL | DB_INIT_TXN | DB_RECOVER | DB_THREAD,
     S_IRUSR | S_IWUSR)) != 0) {
  (void)dbenv->close(dbenv, 0);
  fprintf(stderr, "dbenv->open: %s: %s\n",
      ENV_DIRECTORY, db_strerror(ret));
  exit (1);
 }

 *dbenvp = dbenv;
}
```

After running this initial program, we can use the db_stat utility to display the contents of the environment directory:

```
prompt> db_stat -e -h TXNAPP
3.2.1   Environment version.
120897  Magic number.
```

```
0       Panic value.
1       References.
6       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Mpool Region: 4.
264KB   Size (270336 bytes).
-1      Segment ID.
1       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Log Region: 3.
96KB    Size (98304 bytes).
-1      Segment ID.
3       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Lock Region: 2.
240KB   Size (245760 bytes).
-1      Segment ID.
1       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Txn Region: 5.
8KB     Size (8192 bytes).
-1      Segment ID.
1       Locks granted without waiting.
0       Locks granted after waiting.
```

Opening the databases

Next, we open three databases ("color" and "fruit" and "cats"), in the database environment. Again, our DB database handles are declared to be free-threaded using the DB_THREAD flag, and so may be used by any number of threads we subsequently create.

```
int
main(int argc, char *argv)
{
 extern int optind;
 DB *db_cats, *db_color, *db_fruit;
 DB_ENV *dbenv;
 int ch;

 while ((ch = getopt(argc, argv, "")) != EOF)
  switch (ch) {
  case '?':
  default:
   usage();
  }
```

```
 argc -= optind;
 argv += optind;

 env_dir_create();
 env_open(&dbenv);

 /* Open database: Key is fruit class; Data is specific type. */
 if (db_open(dbenv, &db_fruit, "fruit", 0))
  return (1);

 /* Open database: Key is a color; Data is an integer. */
 if (db_open(dbenv, &db_color, "color", 0))
  return (1);

 /*
  * Open database:
  * Key is a name; Data is: company name, cat breeds.
  */
 if (db_open(dbenv, &db_cats, "cats", 1))
  return (1);

 return (0);
}

int
db_open(DB_ENV *dbenv, DB **dbp, char *name, int dups)
{
 DB *db;
 int ret;

 /* Create the database handle. */
 if ((ret = db_create(&db, dbenv, 0)) != 0) {
  dbenv->err(dbenv, ret, "db_create");
  return (1);
 }

 /* Optionally, turn on duplicate data items. */
 if (dups && (ret = db->set_flags(db, DB_DUP)) != 0) {
  (void)db->close(db, 0);
  dbenv->err(dbenv, ret, "db->set_flags: DB_DUP");
  return (1);
 }

 /*
  * Open a database in the environment:
  * create if it doesn't exist
  * free-threaded handle
  * read/write owner only
  */
```

```
 if ((ret = db->open(db, NULL, name, NULL, DB_BTREE,
     DB_AUTO_COMMIT | DB_CREATE | DB_THREAD, S_IRUSR | S_IWUSR)) != 0) {
  (void)db->close(db, 0);
  dbenv->err(dbenv, ret, "db->open: %s", name);
  return (1);
 }

 *dbp = db;
 return (0);
}
```

After opening the database, we can use the db_stat utility to display information about a database we have created:

```
prompt> db_stat -h TXNAPP -d color
53162   Btree magic number.
8       Btree version number.
Flags:
2       Minimum keys per-page.
8192    Underlying database page size.
1       Number of levels in the tree.
0       Number of unique keys in the tree.
0       Number of data items in the tree.
0       Number of tree internal pages.
0       Number of bytes free in tree internal pages (0% ff).
1       Number of tree leaf pages.
8166    Number of bytes free in tree leaf pages (0.% ff).
0       Number of tree duplicate pages.
0       Number of bytes free in tree duplicate pages (0% ff).
0       Number of tree overflow pages.
0       Number of bytes free in tree overflow pages (0% ff).
0       Number of pages on the free list.
```

The database open must be enclosed within a transaction in order to be recoverable. The transaction will ensure that created files are re-created in recovered environments (or do not appear at all). Additional database operations or operations on other databases can be included in the same transaction, of course. In the simple case, where the open is the only operation in the transaction, an application can set the DB_AUTO_COMMIT flag instead of creating and managing its own transaction handle. The DB_AUTO_COMMIT flag will internally wrap the operation in a transaction, simplifying application code.

The previous example is the simplest case of transaction protection for database open. Obviously, additional database operations can be done in the scope of the same transaction. For example, an application maintaining a list of the databases in a database environment in a well-known file might include an update of the list in the same transaction in which the database is created. Or, an application might create both a primary and secondary database in a single transaction.

DB handles that will later be used for transactionally protected database operations must be opened within a transaction. Specifying a transaction handle to database operations using DB

handles not opened within a transaction will return an error. Similarly, not specifying a transaction handle to database operations that will modify the database, using handles that were opened within a transaction, will also return an error.

Recoverability and deadlock handling

The first reason listed for using transactions was recoverability. Any logical change to a database may require multiple changes to underlying data structures. For example, modifying a record in a Btree may require leaf and internal pages to split, so a single DB->put() method call can potentially require that multiple physical database pages be written. If only some of those pages are written and then the system or application fails, the database is left inconsistent and cannot be used until it has been recovered; that is, until the partially completed changes have been undone.

Write-ahead-logging is the term that describes the underlying implementation that Berkeley DB uses to ensure recoverability. What it means is that before any change is made to a database, information about the change is written to a database log. During recovery, the log is read, and databases are checked to ensure that changes described in the log for committed transactions appear in the database. Changes that appear in the database but are related to aborted or unfinished transactions in the log are undone from the database.

For recoverability after application or system failure, operations that modify the database must be protected by transactions. More specifically, operations are not recoverable unless a transaction is begun and each operation is associated with the transaction via the Berkeley DB interfaces, and then the transaction successfully committed. This is true even if logging is turned on in the database environment.

Here is an example function that updates a record in a database in a transactionally protected manner. The function takes a key and data items as arguments and then attempts to store them into the database.

```
int
main(int argc, char *argv)
{
 extern int optind;
 DB *db_cats, *db_color, *db_fruit;
 DB_ENV *dbenv;
 int ch;

 while ((ch = getopt(argc, argv, "")) != EOF)
  switch (ch) {
  case '?':
  default:
   usage();
  }
 argc -= optind;
 argv += optind;

 env_dir_create();
 env_open(&dbenv);
```

```
 /* Open database: Key is fruit class; Data is specific type. */
 db_open(dbenv, &db_fruit, "fruit", 0);

 /* Open database: Key is a color; Data is an integer. */
 db_open(dbenv, &db_color, "color", 0);

 /*
  * Open database:
  * Key is a name; Data is: company name, cat breeds.
  */
 db_open(dbenv, &db_cats, "cats", 1);

 add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

 return (0);
}

int
add_fruit(DB_ENV *dbenv, DB *db, char *fruit, char *name)
{
 DBT key, data;
 DB_TXN *tid;
 int fail, ret, t_ret;

 /* Initialization. */
 memset(&key, 0, sizeof(key));
 memset(&data, 0, sizeof(data));
 key.data = fruit;
 key.size = strlen(fruit);
 data.data = name;
 data.size = strlen(name);

 for (fail = 0;;) {
  /* Begin the transaction. */
  if ((ret = dbenv->txn_begin(dbenv, NULL, &tid, 0)) != 0) {
   dbenv->err(dbenv, ret, "DB_ENV->txn_begin");
   exit (1);
  }

  /* Store the value. */
  switch (ret = db->put(db, tid, &key, &data, 0)) {
  case 0:
   /* Success: commit the change. */
   if ((ret = tid->commit(tid, 0)) != 0) {
    dbenv->err(dbenv, ret, "DB_TXN->commit");
    exit (1);
   }
   return (0);
```

```
   case DB_LOCK_DEADLOCK:
   default:
    /* 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);
    break;
   }
  }
}
```

Berkeley DB also uses transactions to recover from deadlock. Database operations (that is, any call to a function underlying the handles returned by DB->open() and DB->cursor()) are usually performed on behalf of a unique locker. Transactions can be used to perform multiple calls on behalf of the same locker within a single thread of control. For example, consider the case in which an application uses a cursor scan to locate a record and then the application accesses another other item in the database, based on the key returned by the cursor, without first closing the cursor. If these operations are done using default locker IDs, they may conflict. If the locks are obtained on behalf of a transaction, using the transaction's locker ID instead of the database handle's locker ID, the operations will not conflict.

There is a new error return in this function that you may not have seen before. In transactional (not Concurrent Data Store) applications supporting both readers and writers, or just multiple writers, Berkeley DB functions have an additional possible error return: DB_LOCK_DEADLOCK (page 230). This means two threads of control deadlocked, and the thread receiving the DB_LOCK_DEADLOCK error return has been selected to discard its locks in order to resolve the problem. When an application receives a DB_LOCK_DEADLOCK return, the correct action is to close any cursors involved in the operation and abort any enclosing transaction. In the sample code, any time the DB->put() method returns DB_LOCK_DEADLOCK, DB_TXN->abort() is called (which releases the transaction's Berkeley DB resources and undoes any partial changes to the databases), and then the transaction is retried from the beginning.

There is no requirement that the transaction be attempted again, but that is a common course of action for applications. Applications may want to set an upper bound on the number of times an operation will be retried because some operations on some data sets may simply be unable to succeed. For example, updating all of the pages on a large Web site during prime business hours may simply be impossible because of the high access rate to the database.

The DB_TXN->abort() method is called in error cases other than deadlock. Any time an error occurs, such that a transactionally protected set of operations cannot complete successfully, the transaction must be aborted. While deadlock is by far the most common of these errors, there are other possibilities; for example, running out of disk space for the filesystem. In Berkeley DB transactional applications, there are three classes of error returns: "expected" errors, "unexpected but recoverable" errors, and a single "unrecoverable" error. Expected errors are errors like DB_NOTFOUND (page 230), which indicates that a searched-for key item is not present in the database. Applications may want to explicitly test for and handle this error, or,

in the case where the absence of a key implies the enclosing transaction should fail, simply call DB_TXN->abort(). Unexpected but recoverable errors are errors like DB_LOCK_DEADLOCK (page 230), which indicates that an operation has been selected to resolve a deadlock, or a system error such as EIO, which likely indicates that the filesystem has no available disk space. Applications must immediately call DB_TXN->abort() when these returns occur, as it is not possible to proceed otherwise. The only unrecoverable error is DB_RUNRECOVERY (page 230), which indicates that the system must stop and recovery must be run.

The above code can be simplified in the case of a transaction comprised entirely of a single database put or delete operation, as operations occurring in transactional databases are implicitly transaction protected. For example, in a transactional database, the above code could be more simply written as:

```
for (fail = 0; fail++ <= MAXIMUM_RETRY &&
    (ret = db->put(db, NULL, &key, &data, 0)) == DB_LOCK_DEADLOCK;)
 ;
return (ret == 0 ? 0 : 1);
```

and the underlying transaction would be automatically handled by Berkeley DB.

Programmers should not attempt to enumerate all possible error returns in their software. Instead, they should explicitly handle expected returns and default to aborting the transaction for the rest. It is entirely the choice of the programmer whether to check for DB_RUNRECOVERY (page 230) explicitly or not — attempting new Berkeley DB operations after DB_RUNRECOVERY (page 230) is returned does not worsen the situation. Alternatively, using the DB_ENV->set_event_notify() method to handle an unrecoverable error and simply doing some number of abort-and-retry cycles for any unexpected Berkeley DB or system error in the mainline code often results in the simplest and cleanest application code.

Atomicity

The second reason listed for using transactions was *atomicity*. Atomicity means that multiple operations can be grouped into a single logical entity, that is, other threads of control accessing the database will either see all of the changes or none of the changes. Atomicity is important for applications wanting to update two related databases (for example, a primary database and secondary index) in a single logical action. Or, for an application wanting to update multiple records in one database in a single logical action.

Any number of operations on any number of databases can be included in a single transaction to ensure the atomicity of the operations. There is, however, a trade-off between the number of operations included in a single transaction and both throughput and the possibility of deadlock. The reason for this is because transactions acquire locks throughout their lifetime and do not release the locks until commit or abort time. So, the more operations included in a transaction, the more likely it is that a transaction will block other operations and that deadlock will occur. However, each transaction commit requires a synchronous disk I/O, so grouping multiple operations into a transaction can increase overall throughput. (There is one exception to this: the DB_TXN_WRITE_NOSYNC and DB_TXN_NOSYNC flags cause transactions to exhibit the ACI (atomicity, consistency and isolation) properties, but not D (durability);

avoiding the write and/or synchronous disk I/O on transaction commit greatly increases transaction throughput for some applications.)

When applications do create complex transactions, they often avoid having more than one complex transaction at a time because simple operations like a single DB->put() are unlikely to deadlock with each other or the complex transaction; while multiple complex transactions are likely to deadlock with each other because they will both acquire many locks over their lifetime. Alternatively, complex transactions can be broken up into smaller sets of operations, and each of those sets may be encapsulated in a nested transaction. Because nested transactions may be individually aborted and retried without causing the entire transaction to be aborted, this allows complex transactions to proceed even in the face of heavy contention, repeatedly trying the suboperations until they succeed.

It is also helpful to order operations within a transaction; that is, access the databases and items within the databases in the same order, to the extent possible, in all transactions. Accessing databases and items in different orders greatly increases the likelihood of operations being blocked and failing due to deadlocks.

Isolation

The third reason listed for using transactions was *isolation*. Consider an application suite in which multiple threads of control (multiple processes or threads in one or more processes) are changing the values associated with a key in one or more databases. Specifically, they are taking the current value, incrementing it, and then storing it back into the database.

Such an application requires isolation. Because we want to change a value in the database, we must make sure that after we read it, no other thread of control modifies it. For example, assume that both thread #1 and thread #2 are doing similar operations in the database, where thread #1 is incrementing records by 3, and thread #2 is incrementing records by 5. We want to increment the record by a total of 8. If the operations interleave in the right (well, wrong) order, that is not what will happen:

```
thread #1  read record: the value is 2
thread #2  read record: the value is 2
thread #2  write record + 5 back into the database (new value 7)
thread #1  write record + 3 back into the database (new value 5)
```

As you can see, instead of incrementing the record by a total of 8, we've incremented it only by 3 because thread #1 overwrote thread #2's change. By wrapping the operations in transactions, we ensure that this cannot happen. In a transaction, when the first thread reads the record, locks are acquired that will not be released until the transaction finishes, guaranteeing that all writers will block, waiting for the first thread's transaction to complete (or to be aborted).

Here is an example function that does transaction-protected increments on database records to ensure isolation:

```
int
main(int argc, char *argv)
{
 extern int optind;
```

```
 DB *db_cats, *db_color, *db_fruit;
 DB_ENV *dbenv;
 int ch;

 while ((ch = getopt(argc, argv, "")) != EOF)
  switch (ch) {
  case '?':
  default:
   usage();
  }
 argc -= optind;
 argv += optind;

 env_dir_create();
 env_open(&dbenv);

 /* Open database: Key is fruit class; Data is specific type. */
 db_open(dbenv, &db_fruit, "fruit", 0);

 /* Open database: Key is a color; Data is an integer. */
 db_open(dbenv, &db_color, "color", 0);

 /*
  * Open database:
  * Key is a name; Data is: company name, cat breeds.
  */
 db_open(dbenv, &db_cats, "cats", 1);

 add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

 add_color(dbenv, db_color, "blue", 0);
 add_color(dbenv, db_color, "blue", 3);

 return (0);
}

int
add_color(DB_ENV *dbenv, DB *dbp, char *color, int increment)
{
 DBT key, data;
 DB_TXN *tid;
 int fail, original, ret, t_ret;
 char buf64;

 /* Initialization. */
 memset(&key, 0, sizeof(key));
 key.data = color;
 key.size = strlen(color);
 memset(&data, 0, sizeof(data));
```

```
data.flags = DB_DBT_MALLOC;

for (fail = 0;;) {
 /* Begin the transaction. */
 if ((ret = dbenv->txn_begin(dbenv, NULL, &tid, 0)) != 0) {
  dbenv->err(dbenv, ret, "DB_ENV->txn_begin");
  exit (1);
 }

 /*
  * Get the key.  If it exists, we increment the value.  If it
  * doesn't exist, we create it.
  */
 switch (ret = dbp->get(dbp, tid, &key, &data, DB_RMW)) {
 case 0:
  original = atoi(data.data);
  break;
 case DB_LOCK_DEADLOCK:
 default:
  /* 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;
 case DB_NOTFOUND:
  original = 0;
  break;
 }
 if (data.data != NULL)
  free(data.data);

 /* Create the new data item. */
 (void)snprintf(buf, sizeof(buf), "%d", original + increment);
 data.data = buf;
 data.size = strlen(buf) + 1;

 /* Store the new value. */
 switch (ret = dbp->put(dbp, tid, &key, &data, 0)) {
 case 0:
  /* Success: commit the change. */
  if ((ret = tid->commit(tid, 0)) != 0) {
   dbenv->err(dbenv, ret, "DB_TXN->commit");
   exit (1);
  }
  return (0);
 case DB_LOCK_DEADLOCK:
```

```
  default:
   /* 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);
   break;
  }
 }
}
```

The DB_RMW flag in the DB->get() call specifies a write lock should be acquired on the key/data pair, instead of the more obvious read lock. We do this because the application expects to write the key/data pair in a subsequent operation, and the transaction is much more likely to deadlock if we first obtain a read lock and subsequently a write lock, than if we obtain the write lock initially.

Degrees of isolation

Transactions can be isolated from each other to different degrees. *Serializable* provides the most isolation, and means that, for the life of the transaction, every time a thread of control reads a data item, it will be unchanged from its previous value (assuming, of course, the thread of control does not itself modify the item). By default, Berkeley DB enforces serializability whenever database reads are wrapped in transactions. This is also known as *degree 3 isolation*.

Most applications do not need to enclose all reads in transactions, and when possible, transactionally protected reads at serializable isolation should be avoided as they can cause performance problems. For example, a serializable cursor sequentially reading each key/data pair in a database, will acquire a read lock on most of the pages in the database and so will gradually block all write operations on the databases until the transaction commits or aborts. Note, however, that if there are update transactions present in the application, the read operations must still use locking, and must be prepared to repeat any operation (possibly closing and reopening a cursor) that fails with a return value of DB_LOCK_DEADLOCK (page 230). Applications that need repeatable reads are ones that require the ability to repeatedly access a data item knowing that it will not have changed (for example, an operation modifying a data item based on its existing value).

Snapshot isolation also guarantees repeatable reads, but avoids read locks by using multiversion concurrency control (MVCC). This makes update operations more expensive, because they have to allocate space for new versions of pages in cache and make copies, but avoiding read locks can significantly increase throughput for many applications. Snapshot isolation is discussed in detail below.

A transaction may only require *cursor stability*, that is only be guaranteed that cursors see committed data that does not change so long as it is addressed by the cursor, but may change before the reading transaction completes. This is also called *degree 2 isolation*. Berkeley DB

provides this level of isolation when a transaction is started with the DB_READ_COMMITTED flag. This flag may also be specified when opening a cursor within a fully isolated transaction.

Berkeley DB optionally supports reading uncommitted data; that is, read operations may request data which has been modified but not yet committed by another transaction. This is also called *degree 1 isolation*. This is done by first specifying the DB_READ_UNCOMMITTED flag when opening the underlying database, and then specifying the DB_READ_UNCOMMITTED flag when beginning a transaction, opening a cursor, or performing a read operation. The advantage of using DB_READ_UNCOMMITTED is that read operations will not block when another transaction holds a write lock on the requested data; the disadvantage is that read operations may return data that will disappear should the transaction holding the write lock abort.

Snapshot Isolation

To make use of snapshot isolation, databases must first be configured for multiversion access by calling DB->open() with the DB_MULTIVERSION flag. Then transactions or cursors must be configured with the DB_TXN_SNAPSHOT flag.

When configuring an environment for snapshot isolation, it is important to realize that having multiple versions of pages in cache means that the working set will take up more of the cache. As a result, snapshot isolation is best suited for use with larger cache sizes.

If the cache becomes full of page copies before the old copies can be discarded, additional I/O will occur as pages are written to temporary "freezer" files. This can substantially reduce throughput, and should be avoided if possible by configuring a large cache and keeping snapshot isolation transactions short. The amount of cache required to avoid freezing buffers can be estimated by taking a checkpoint followed by a call to DB_ENV->log_archive(). The amount of cache required is approximately double the size of logs that remains.

The environment should also be configured for sufficient transactions using DB_ENV->set_tx_max(). The maximum number of transactions needs to include all transactions executed concurrently by the application plus all cursors configured for snapshot isolation. Further, the transactions are retained until the last page they created is evicted from cache, so in the extreme case, an additional transaction may be needed for each page in the cache. Note that cache sizes under 500MB are increased by 25%, so the calculation of number of pages needs to take this into account.

So when *should* applications use snapshot isolation?

- There is a large cache relative to size of updates performed by concurrent transactions; and
- Read/write contention is limiting the throughput of the application; or
- The application is all or mostly read-only, and contention for the lock manager mutex is limiting throughput.

The simplest way to take advantage of snapshot isolation is for queries: keep update transactions using full read/write locking and set DB_TXN_SNAPSHOT on read-only transactions or cursors. This should minimize blocking of snapshot isolation transactions and will avoid introducing new DB_LOCK_DEADLOCK (page 230) errors.

If the application has update transactions which read many items and only update a small set (for example, scanning until a desired record is found, then modifying it), throughput may be improved by running some updates at snapshot isolation as well.

Transactional cursors

Berkeley DB cursors may be used inside a transaction, exactly as any other DB method. The enclosing transaction ID must be specified when the cursor is created, but it does not then need to be further specified on operations performed using the cursor. One important point to remember is that a cursor **must be closed** before the enclosing transaction is committed or aborted.

The following code fragment uses a cursor to store a new key in the cats database with four associated data items. The key is a name. The data items are a company name and a list of the breeds of cat owned. Each of the data entries is stored as a duplicate data item. In this example, transactions are necessary to ensure that either all or none of the data items appear in case of system or application failure.

```
int
main(int argc, char *argv)
{
 extern int optind;
 DB *db_cats, *db_color, *db_fruit;
 DB_ENV *dbenv;
 int ch;

 while ((ch = getopt(argc, argv, "")) != EOF)
  switch (ch) {
  case '?':
  default:
   usage();
  }
 argc -= optind;
 argv += optind;

 env_dir_create();
 env_open(&dbenv);

 /* Open database: Key is fruit class; Data is specific type. */
 db_open(dbenv, &db_fruit, "fruit", 0);

 /* Open database: Key is a color; Data is an integer. */
 db_open(dbenv, &db_color, "color", 0);

 /*
  * Open database:
  * Key is a name; Data is: company name, cat breeds.
  */
 db_open(dbenv, &db_cats, "cats", 1);
```

```
 add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

 add_color(dbenv, db_color, "blue", 0);
 add_color(dbenv, db_color, "blue", 3);

 add_cat(dbenv, db_cats,
  "Amy Adams",
  "Oracle",
  "abyssinian",
  "bengal",
  "chartreaux",
  NULL);

 return (0);
}

int
add_cat(DB_ENV *dbenv, DB *db, char *name, ...)
{
 va_list ap;
 DBC *dbc;
 DBT key, data;
 DB_TXN *tid;
 int fail, ret, t_ret;
 char *s;

 /* Initialization. */
 fail = 0;

 memset(&key, 0, sizeof(key));
 memset(&data, 0, sizeof(data));
 key.data = name;
 key.size = strlen(name);

retry: /* Begin the transaction. */
 if ((ret = dbenv->txn_begin(dbenv, NULL, &tid, 0)) != 0) {
  dbenv->err(dbenv, ret, "DB_ENV->txn_begin");
  exit (1);
 }

 /* Delete any previously existing item. */
 switch (ret = db->del(db, tid, &key, 0)) {
 case 0:
 case DB_NOTFOUND:
  break;
 case DB_LOCK_DEADLOCK:
 default:
  /* 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);
 goto retry;
}

/* Create a cursor. */
if ((ret = db->cursor(db, tid, &dbc, 0)) != 0) {
 dbenv->err(dbenv, ret, "db->cursor");
 exit (1);
}

/* Append the items, in order. */
va_start(ap, name);
while ((s = va_arg(ap, char *)) != NULL) {
 data.data = s;
 data.size = strlen(s);
 switch (ret = dbc->c_put(dbc, &key, &data, DB_KEYLAST)) {
 case 0:
  break;
 case DB_LOCK_DEADLOCK:
 default:
  va_end(ap);

  /* Retry the operation. */
  if ((t_ret = dbc->c_close(dbc)) != 0) {
   dbenv->err(
       dbenv, t_ret, "dbc->c_close");
   exit (1);
  }
  if ((t_ret = tid->abort(tid)) != 0) {
   dbenv->err(dbenv, t_ret, "DB_TXN->abort");
   exit (1);
  }
  if (fail++ == MAXIMUM_RETRY)
   return (ret);
  goto retry;
 }
}
va_end(ap);

/* Success: commit the change. */
if ((ret = dbc->c_close(dbc)) != 0) {
 dbenv->err(dbenv, ret, "dbc->c_close");
 exit (1);
}
```

```
 if ((ret = tid->commit(tid, 0)) != 0) {
  dbenv->err(dbenv, ret, "DB_TXN->commit");
  exit (1);
 }
 return (0);
}
```

Nested transactions

Berkeley DB provides support for nested transactions. Nested transactions allow an application to decompose a large or long-running transaction into smaller units that may be independently aborted.

Normally, when beginning a transaction, the application will pass a NULL value for the parent argument to DB_ENV->txn_begin(). If, however, the parent argument is a TXN handle, the newly created transaction will be treated as a nested transaction within the parent. Transactions may nest arbitrarily deeply. For the purposes of this discussion, transactions created with a parent identifier will be called *child transactions*.

Once a transaction becomes a parent, as long as any of its child transactions are unresolved (that is, they have neither committed nor aborted), the parent may not issue any Berkeley DB calls except to begin more child transactions, or to commit or abort. For example, it may not issue any access method or cursor calls. After all of a parent's children have committed or aborted, the parent may again request operations on its own behalf.

The semantics of nested transactions are as follows. When a child transaction is begun, it inherits all the locks of its parent. This means that the child will never block waiting on a lock held by its parent. Further, locks held by two children of the same parent will also conflict. To make this concrete, consider the following set of transactions and lock acquisitions.

Transaction T1 is the parent transaction. It acquires a write lock on item A and then begins two child transactions: C1 and C2. C1 also wants to acquire a write lock on A; this succeeds. If C2 attempts to acquire a write lock on A, it will block until C1 releases the lock, at which point it will succeed. Now, let's say that C1 acquires a write lock on B. If C2 now attempts to obtain a lock on B, it will block. However, let's now assume that C1 commits. Its locks are anti-inherited, which means they are given to T1, so T1 will now hold a lock on B. At this point, C2 would be unblocked and would then acquire a lock on B.

Child transactions are entirely subservient to their parent transaction. They may abort, undoing their operations regardless of the eventual fate of the parent. However, even if a child transaction commits, if its parent transaction is eventually aborted, the child's changes are undone and the child's transaction is effectively aborted. Any child transactions that are not yet resolved when the parent commits or aborts are resolved based on the parent's resolution -- committing if the parent commits and aborting if the parent aborts. Any child transactions that are not yet resolved when the parent prepares are also prepared.

Environment infrastructure

When building transactional applications, it is usually necessary to build an administrative infrastructure around the database environment. There are five components to this infrastructure, and each is supported by the Berkeley DB package in two different ways: a standalone utility and one or more library interfaces.

- Deadlock detection: db_deadlock utility, DB_ENV->lock_detect(), DB_ENV->set_lk_detect()
- Checkpoints: the db_checkpoint utility, DB_ENV->txn_checkpoint()
- Database and log file archival: the db_archive utility, DB_ENV->log_archive()
- Log file removal: db_archive utility, DB_ENV->log_archive()
- Recovery procedures: db_recover utility, DB_ENV->open()

When writing multithreaded server applications and/or applications intended for download from the Web, it is usually simpler to create local threads that are responsible for administration of the database environment as scheduling is often simpler in a single-process model, and only a single binary need be installed and run. However, the supplied utilities can be generally useful tools even when the application is responsible for doing its own administration because applications rarely offer external interfaces to database administration. The utilities are required when programming to a Berkeley DB scripting interface because the scripting APIs do not always offer interfaces to the administrative functionality.

Deadlock detection

The first component of the infrastructure, *deadlock detection*, is not so much a requirement specific to transaction-protected applications, but instead is necessary for almost all applications in which more than a single thread of control will be accessing the database at one time. Even when Berkeley DB automatically handles database locking, it is normally possible for deadlock to occur. Because the underlying database access methods may update multiple pages during a single Berkeley DB API call, deadlock is possible even when threads of control are making only single update calls into the database. The exception to this rule is when all the threads of control accessing the database are read-only or when the Berkeley DB Concurrent Data Store product is used; the Berkeley DB Concurrent Data Store product guarantees deadlock-free operation at the expense of reduced concurrency.

When the deadlock occurs, two (or more) threads of control each request additional locks that can never be granted because one of the threads of control waiting holds the requested resource. For example, consider two processes: A and B. Let's say that A obtains a write lock on item X, and B obtains a write lock on item Y. Then, A requests a lock on Y, and B requests a lock on X. A will wait until resource Y becomes available and B will wait until resource X becomes available. Unfortunately, because both A and B are waiting, neither will release the locks they hold and neither will ever obtain the resource on which it is waiting. For another example, consider two transactions, A and B, each of which may want to modify item X. Assume that transaction A obtains a read lock on X and confirms that a modification is needed. Then it is descheduled and the thread containing transaction B runs. At that time, transaction B obtains

a read lock on X and confirms that it also wants to make a modification. Both transactions A and B will block when they attempt to upgrade their read locks to write locks because the other already holds a read lock. This is a deadlock. Transaction A cannot make forward progress until Transaction B releases its read lock on X, but Transaction B cannot make forward progress until Transaction A releases its read lock on X.

In order to detect that deadlock has happened, a separate process or thread must review the locks currently held in the database. If deadlock has occurred, a victim must be selected, and that victim will then return the error DB_LOCK_DEADLOCK (page 230) from whatever Berkeley DB call it was making. Berkeley DB provides the db_deadlock utility that can be used to perform this deadlock detection. Alternatively, applications can create their own deadlock utility or thread using the underlying DB_ENV->lock_detect() function, or specify that Berkeley DB run the deadlock detector internally whenever there is a conflict over a lock (see DB_ENV->set_lk_detect() for more information). The following code fragment does the latter:

```
void
env_open(DB_ENV **dbenvp)
{
 DB_ENV *dbenv;
 int ret;

 /* Create the environment handle. */
 if ((ret = db_env_create(&dbenv, 0)) != 0) {
  fprintf(stderr,
      "txnapp: db_env_create: %s\n", db_strerror(ret));
  exit (1);
 }

 /* Set up error handling. */
 dbenv->set_errpfx(dbenv, "txnapp");
 dbenv->set_errfile(dbenv, stderr);

 /* Do deadlock detection internally. */
 if ((ret = dbenv->set_lk_detect(dbenv, DB_LOCK_DEFAULT)) != 0) {
  dbenv->err(dbenv, ret, "set_lk_detect: DB_LOCK_DEFAULT");
  exit (1);
 }

 /*
  * Open a transactional environment:
  * create if it doesn't exist
  * free-threaded handle
  * run recovery
  * read/write owner only
  */
 if ((ret = dbenv->open(dbenv, ENV_DIRECTORY,
     DB_CREATE | DB_INIT_LOCK | DB_INIT_LOG |
     DB_INIT_MPOOL | DB_INIT_TXN | DB_RECOVER | DB_THREAD,
     S_IRUSR | S_IWUSR)) != 0) {
```

```
  dbenv->err(dbenv, ret, "dbenv->open: %s", ENV_DIRECTORY);
  exit (1);
 }

 *dbenvp = dbenv;
}
```

Deciding how often to run the deadlock detector and which of the deadlocked transactions will be forced to abort when the deadlock is detected is a common tuning parameter for Berkeley DB applications.

Checkpoints

The second component of the infrastructure is performing checkpoints of the log files. Performing checkpoints is necessary for two reasons.

First, you may be able to remove Berkeley DB log files from your database environment after a checkpoint. Change records are written into the log files when databases are modified, but the actual changes to the database are not necessarily written to disk. When a checkpoint is performed, changes to the database are written into the backing database file. Once the database pages are written, log files can be archived and removed from the database environment because they will never be needed for anything other than catastrophic failure. (Log files which are involved in active transactions may not be removed, and there must always be at least one log file in the database environment.)

The second reason to perform checkpoints is because checkpoint frequency is inversely proportional to the amount of time it takes to run database recovery after a system or application failure. This is because recovery after failure has to redo or undo changes only since the last checkpoint, as changes before the checkpoint have all been flushed to the databases.

Berkeley DB provides the db_checkpoint utility, which can be used to perform checkpoints. Alternatively, applications can write their own checkpoint thread using the underlying DB_ENV->txn_checkpoint() function. The following code fragment checkpoints the database environment every 60 seconds:

```
int
main(int argc, char *argv)
{
 extern int optind;
 DB *db_cats, *db_color, *db_fruit;
 DB_ENV *dbenv;
 pthread_t ptid;
 int ch;

 while ((ch = getopt(argc, argv, "")) != EOF)
  switch (ch) {
  case '?':
  default:
   usage();
```

```
 }
argc -= optind;
argv += optind;

env_dir_create();
env_open(&dbenv);

/* Start a checkpoint thread. */
if ((errno = pthread_create(
    &ptid, NULL, checkpoint_thread, (void *)dbenv)) != 0) {
 fprintf(stderr,
     "txnapp: failed spawning checkpoint thread: %s\n",
     strerror(errno));
 exit (1);
}

/* Open database: Key is fruit class; Data is specific type. */
db_open(dbenv, &db_fruit, "fruit", 0);

/* Open database: Key is a color; Data is an integer. */
db_open(dbenv, &db_color, "color", 0);

/*
 * Open database:
 * Key is a name; Data is: company name, cat breeds.
 */
db_open(dbenv, &db_cats, "cats", 1);

add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

add_color(dbenv, db_color, "blue", 0);
add_color(dbenv, db_color, "blue", 3);

add_cat(dbenv, db_cats,
 "Amy Adams",
 "Oracle",
 "abyssinian",
 "bengal",
 "chartreaux",
 NULL);

return (0);
}

void *
checkpoint_thread(void *arg)
{
 DB_ENV *dbenv;
 int ret;
```

```
 dbenv = arg;
 dbenv->errx(dbenv, "Checkpoint thread: %lu", (u_long)pthread_self());

 /* Checkpoint once a minute. */
 for (;; sleep(60))
  if ((ret = dbenv->txn_checkpoint(dbenv, 0, 0, 0)) != 0) {
   dbenv->err(dbenv, ret, "checkpoint thread");
   exit (1);
  }

 /* NOTREACHED */
}
```

Because checkpoints can be quite expensive, choosing how often to perform a checkpoint is a common tuning parameter for Berkeley DB applications.

Database and log file archival

The third component of the administrative infrastructure, archival for catastrophic recovery, concerns the recoverability of the database in the face of catastrophic failure. Recovery after catastrophic failure is intended to minimize data loss when physical hardware has been destroyed -- for example, loss of a disk that contains databases or log files. Although the application may still experience data loss in this case, it is possible to minimize it.

First, you may want to periodically create snapshots (that is, backups) of your databases to make it possible to recover from catastrophic failure. These snapshots are either a standard backup, which creates a consistent picture of the databases as of a single instant in time; or an on-line backup (also known as a *hot* backup), which creates a consistent picture of the databases as of an unspecified instant during the period of time when the snapshot was made. The advantage of a hot backup is that applications may continue to read and write the databases while the snapshot is being taken. The disadvantage of a hot backup is that more information must be archived, and recovery based on a hot backup is to an unspecified time between the start of the backup and when the backup is completed.

Second, after taking a snapshot, you should periodically archive the log files being created in the environment. It is often helpful to think of database archival in terms of full and incremental filesystem backups. A snapshot is a full backup, whereas the periodic archival of the current log files is an incremental backup. For example, it might be reasonable to take a full snapshot of a database environment weekly or monthly, and archive additional log files daily. Using both the snapshot and the log files, a catastrophic crash at any time can be recovered to the time of the most recent log archival; a time long after the original snapshot.

To create a standard backup of your database that can be used to recover from catastrophic failure, take the following steps:

1. Commit or abort all ongoing transactions.

2. Stop writing your databases until the backup has completed. Read-only operations are permitted, but no write operations and no filesystem operations may be performed (for example, the DB_ENV->remove() and DB->open() methods may not be called).

3. Force an environment checkpoint (see the db_checkpoint utility for more information).

4. Run the db_archive utility with option **-s** to identify all the database data files, and copy them to a backup device such as CD-ROM, alternate disk, or tape.

   If the database files are stored in a separate directory from the other Berkeley DB files, it may be simpler to archive the directory itself instead of the individual files (see DB_ENV->set_data_dir() for additional information). **Note: if any of the database files did not have an open DB handle during the lifetime of the current log files, the db_archive utility will not list them in its output!** This is another reason it may be simpler to use a separate database file directory and archive the entire directory instead of archiving only the files listed by the db_archive utility.

5. Run the db_archive utility with option **-l** to identify all the log files, and copy the last one (that is, the one with the highest number) to a backup device such as CD-ROM, alternate disk, or tape.

To create a *hot* backup of your database that can be used to recover from catastrophic failure, take the following steps:

1. Archive your databases, as described in the previous step #4. You do not have to halt ongoing transactions or force a checkpoint. As this is a hot backup, and the databases may be modified during the copy, the utility you use to copy the databases must read database pages atomically (as described by Berkeley DB recoverability (page 180)).

2. Archive **all** of the log files. The order of these two operations is required, and the database files must be archived **before** the log files. This means that if the database files and log files are in the same directory, you cannot simply archive the directory; you must make sure that the correct order of archival is maintained.

   To archive your log files, run the db_archive utility using the **-l** option to identify all the database log files, and copy them to your backup media. If the database log files are stored in a separate directory from the other database files, it may be simpler to archive the directory itself instead of the individual files (see the DB_ENV->set_lg_dir() method for more information).

To minimize the archival space needed for log files when doing a hot backup, run db_archive to identify those log files which are not in use. Log files which are not in use do not need to be included when creating a hot backup, and you can discard them or move them aside for use with previous backups (whichever is appropriate), before beginning the hot backup.

After completing one of these two sets of steps, the database environment can be recovered from catastrophic failure (see Recovery procedures (page 176) for more information).

For an example of a hot backup implementation in the Berkeley DB distribution, see the source code for the db_hotbackup utility.

To update either a hot or cold backup so that recovery from catastrophic failure is possible to a new point in time, repeat step #2 under the hot backup instructions and archive **all** of the log files in the database environment. Each time both the database and log files are copied to backup media, you may discard all previous database snapshots and saved log files. Archiving additional log files does not allow you to discard either previous database snapshots or log files. Generally, updating a backup must be integrated with the application's log file removal procedures.

The time to restore from catastrophic failure is a function of the number of log records that have been written since the snapshot was originally created. Perhaps more importantly, the more separate pieces of backup media you use, the more likely it is that you will have a problem reading from one of them. For these reasons, it is often best to make snapshots on a regular basis.

Obviously, the reliability of your archive media will affect the safety of your data. For archival safety, ensure that you have multiple copies of your database backups, verify that your archival media is error-free and readable, and that copies of your backups are stored offsite!

The functionality provided by the db_archive utility is also available directly from the Berkeley DB library. The following code fragment prints out a list of log and database files that need to be archived:

```
void
log_archlist(DB_ENV *dbenv)
{
 int ret;
 char **begin, **list;

 /* Get the list of database files. */
 if ((ret = dbenv->log_archive(dbenv,
     &list, DB_ARCH_ABS | DB_ARCH_DATA)) != 0) {
  dbenv->err(dbenv, ret, "DB_ENV->log_archive: DB_ARCH_DATA");
  exit (1);
 }
 if (list != NULL) {
  for (begin = list; *list != NULL; ++list)
   printf("database file: %s\n", *list);
  free (begin);
 }

 /* Get the list of log files. */
 if ((ret = dbenv->log_archive(dbenv,
     &list, DB_ARCH_ABS | DB_ARCH_LOG)) != 0) {
  dbenv->err(dbenv, ret, "DB_ENV->log_archive: DB_ARCH_LOG");
  exit (1);
 }
 if (list != NULL) {
  for (begin = list; *list != NULL; ++list)
   printf("log file: %s\n", *list);
```

```
    free (begin);
  }
}
```

Log file removal

The fourth component of the infrastructure, log file removal, concerns the ongoing disk consumption of the database log files. Depending on the rate at which the application writes to the databases and the available disk space, the number of log files may increase quickly enough so that disk space will be a resource problem. For this reason, you will periodically want to remove log files in order to conserve disk space. This procedure is distinct from database and log file archival for catastrophic recovery, and you cannot remove the current log files simply because you have created a database snapshot or copied log files to archival media.

Log files may be removed at any time, as long as:

- the log file is not involved in an active transaction.
- a checkpoint has been written subsequent to the log file's creation.
- the log file is not the only log file in the environment.

If you are preparing for catastrophic failure, you will want to copy the log files to archival media before you remove them as described in Database and log file archival (page 173).

If you are not preparing for catastrophic failure, any one of the following methods can be used to remove log files:

1. Run the standalone db_archive utility with the **-d** option, to remove any log files that are no longer needed at the time the command is executed.
2. Call the DB_ENV->log_archive() method from the application, with the DB_ARCH_REMOVE flag, to remove any log files that are no longer needed at the time the call is made.
3. Call the DB_ENV->log_set_config() method from the application, with the DB_LOG_AUTO_REMOVE flag, to remove any log files that are no longer needed on an ongoing basis. With this configuration, Berkeley DB will automatically remove log files, and the application will not have an opportunity to copy the log files to backup media.

Recovery procedures

The fifth component of the infrastructure, recovery procedures, concerns the recoverability of the database. After any application or system failure, there are two possible approaches to database recovery:

1. There is no need for recoverability, and all databases can be re-created from scratch. Although these applications may still need transaction protection for other reasons, recovery usually consists of removing the Berkeley DB environment home directory and all files it contains, and then restarting the application. Such an application may use the DB_TXN_NOT_DURABLE flag to avoid writing log records.

2. It is necessary to recover information after system or application failure. In this case, recovery processing must be performed on any database environments that were active at the time of the failure. Recovery processing involves running the db_recover utility or calling the DB_ENV->open() method with the DB_RECOVER or DB_RECOVER_FATAL flags.

   During recovery processing, all database changes made by aborted or unfinished transactions are undone, and all database changes made by committed transactions are redone, as necessary. Database applications must not be restarted until recovery completes. After recovery finishes, the environment is properly initialized so that applications may be restarted.

If performing recovery, there are two types of recovery processing: *normal* and *catastrophic*. Which you choose depends on the source for the database and log files you are using to recover.

If up-to-the-minute database and log files are accessible on a stable filesystem, normal recovery is sufficient. Run the db_recover utility or call the DB_ENV->open() method specifying the DB_RECOVER flag. However, the normal recovery case **never** includes recovery using hot backups of the database environment. For example, you cannot perform a hot backup of databases and log files, restore the backup and then run normal recovery — you must always run catastrophic recovery when using hot backups.

If the database or log files have been destroyed or corrupted, or normal recovery fails, catastrophic recovery is required. For example, catastrophic failure includes the case where the disk drive on which the database or log files are stored has been physically destroyed, or when the underlying filesystem is corrupted and the operating system's normal filesystem checking procedures cannot bring that filesystem to a consistent state. This is often difficult to detect, and a common sign of the need for catastrophic recovery is when normal Berkeley DB recovery procedures fail, or when checksum errors are displayed during normal database procedures. When catastrophic recovery is necessary, take the following steps:

1. Restore the most recent snapshots of the database and log files from the backup media into the directory where recovery will be performed.

2. If any log files were archived since the last snapshot was made, they should be restored into the directory where recovery will be performed.

   If any log files are available from the database environment that failed (for example, the disk holding the database files crashed, but the disk holding the log files is fine), those log files should be copied into the directory where recovery will be performed.

   Be sure to restore all log files in the order they were written. The order is important because it's possible the same log file appears on multiple backups, and you want to run recovery using the most recent version of each log file.

3. Run the db_recover utility, specifying its **-c** option; or call the DB_ENV->open() method, specifying the DB_RECOVER_FATAL flag. The catastrophic recovery process will review the logs and database files to bring the environment databases to a consistent state as of the time of the last uncorrupted log file that is found. It is important to realize that only transactions committed before that date will appear in the databases.

It is possible to re-create the database in a location different from the original by specifying appropriate pathnames to the **-h** option of the db_recover utility. In order for this to work properly, it is important that your application refer to files by names relative to the database home directory or the pathname(s) specified in calls to DB_ENV->set_data_dir(), instead of using full pathnames.

Hot failover

For some applications, it may be useful to periodically snapshot the database environment for use as a hot failover should the primary system fail. The following steps can be taken to keep a backup environment in close synchrony with an active environment. The active environment is entirely unaffected by these procedures, and both read and write operations are allowed during all steps described here.

1. Run the db_archive utility with the **-s** option in the active environment to identify all of the active environment's database files, and copy them to the backup directory.

   If the database files are stored in a separate directory from the other Berkeley DB files, it will be simpler (and much faster!) to copy the directory itself instead of the individual files (see DB_ENV->set_data_dir() for additional information). **Note: if any of the database files did not have an open DB handle during the lifetime of the current log files, the db_archive utility will not list them in its output!** This is another reason it may be simpler to use a separate database file directory and copy the entire directory instead of archiving only the files listed by the db_archive utility.

2. Remove all existing log files from the backup directory.

3. Run the db_archive utility with the **-l** option in the active environment to identify all of the active environment's log files, and copy them to the backup directory.

4. Run the db_recover utility with the **-c** option in the backup directory to catastrophically recover the copied environment.

Steps 2, 3 and 4 may be repeated as often as you like. If Step 1 (the initial copy of the database files) is repeated, then Steps 2, 3 and 4 **must** be performed at least once in order to ensure a consistent database environment snapshot.

These procedures must be integrated with your other archival procedures, of course. If you are periodically removing log files from your active environment, you must be sure to copy them to the backup directory before removing them from the active directory. Not copying a log file to the backup directory and subsequently running recovery with it present may leave the backup snapshot of the environment corrupted. A simple way to ensure this never happens is to archive the log files in Step 2 as you remove them from the backup directory, and move inactive log files from your active environment into your backup directory (rather than copying them), in Step 3. The following steps describe this procedure in more detail:

1. Run the db_archive utility with the **-s** option in the active environment to identify all of the active environment's database files, and copy them to the backup directory.

2. Archive all existing log files from the backup directory, moving them to a backup device such as CD-ROM, alternate disk, or tape.

3. Run the db_archive utility (without any option) in the active environment to identify all of the log files in the active environment that are no longer in use, and **move** them to the backup directory.

4. Run the db_archive utility with the **-l** option in the active environment to identify all of the remaining log files in the active environment, and **copy** the log files to the backup directory.

5. Run the db_recover utility with the **-c** option in the backup directory to catastrophically recover the copied environment.

As before, steps 2, 3, 4 and 5 may be repeated as often as you like. If Step 1 (the initial copy of the database files) is repeated, then Steps 2 through 5 **must** be performed at least once in order to ensure a consistent database environment snapshot.

For an example of a hot backup implementation in the Berkeley DB distribution, see the source code for the db_hotbackup utility.

Recovery and filesystem operations

The Berkeley DB API supports creating, removing and renaming files. Creating files is supported by the DB->open() method. Removing files is supported by the DB_ENV->dbremove() and DB->remove() methods. Renaming files is supported by the DB_ENV->dbrename() and DB->rename() methods. (There are two methods for removing and renaming files because one of the methods is transactionally protected and one is not.)

Berkeley DB does not permit specifying the DB_TRUNCATE flag when opening a file in a transaction-protected environment. This is an implicit file deletion, but one that does not always require the same operating system file permissions as deleting and creating a file do.

If you have changed the name of a file or deleted it outside of the Berkeley DB library (for example, you explicitly removed a file using your normal operating system utilities), then it is possible that recovery will not be able to find a database to which the log refers. In this case, the db_recover utility will produce a warning message, saying it was unable to locate a file it expected to find. This message is only a warning because the file may have been subsequently deleted as part of normal database operations before the failure occurred, so is not necessarily a problem.

Generally, any filesystem operations that are performed outside the Berkeley DB interface should be performed at the same time as making a snapshot of the database. To perform filesystem operations correctly, do the following:

1. Cleanly shut down database operations.

   To shut down database operations cleanly, all applications accessing the database environment must be shut down and a transaction checkpoint must be taken. If the applications are not implemented so they can be shut down gracefully (that is, closing all

references to the database environment), recovery must be performed after all applications have been killed to ensure that the underlying databases are consistent on disk.

2. Perform the filesystem operations; for example, remove or rename one or more files.

3. Make an archival snapshot of the database.

   Although this step is not strictly necessary, it is strongly recommended. If this step is not performed, recovery from catastrophic failure will require that recovery first be performed up to the time of the filesystem operations, the filesystem operations be redone, and then recovery be performed from the filesystem operations forward.

4. Restart the database applications.

Berkeley DB recoverability

Berkeley DB recovery is based on write-ahead logging. This means that when a change is made to a database page, a description of the change is written into a log file. This description in the log file is guaranteed to be written to stable storage before the database pages that were changed are written to stable storage. This is the fundamental feature of the logging system that makes durability and rollback work.

If the application or system crashes, the log is reviewed during recovery. Any database changes described in the log that were part of committed transactions and that were never written to the actual database itself are written to the database as part of recovery. Any database changes described in the log that were never committed and that were written to the actual database itself are backed-out of the database as part of recovery. This design allows the database to be written lazily, and only blocks from the log file have to be forced to disk as part of transaction commit.

There are two interfaces that are a concern when considering Berkeley DB recoverability:

1. The interface between Berkeley DB and the operating system/filesystem.

2. The interface between the operating system/filesystem and the underlying stable storage hardware.

Berkeley DB uses the operating system interfaces and its underlying filesystem when writing its files. This means that Berkeley DB can fail if the underlying filesystem fails in some unrecoverable way. Otherwise, the interface requirements here are simple: The system call that Berkeley DB uses to flush data to disk (normally fsync or fdatasync), must guarantee that all the information necessary for a file's recoverability has been written to stable storage before it returns to Berkeley DB, and that no possible application or system crash can cause that file to be unrecoverable.

In addition, Berkeley DB implicitly uses the interface between the operating system and the underlying hardware. The interface requirements here are not as simple.

First, it is necessary to consider the underlying page size of the Berkeley DB databases. The Berkeley DB library performs all database writes using the page size specified by the application,

and Berkeley DB assumes pages are written atomically. This means that if the operating system performs filesystem I/O in blocks of different sizes than the database page size, it may increase the possibility for database corruption. For example, assume that Berkeley DB is writing 32KB pages for a database, and the operating system does filesystem I/O in 16KB blocks. If the operating system writes the first 16KB of the database page successfully, but crashes before being able to write the second 16KB of the database, the database has been corrupted and this corruption may or may not be detected during recovery. For this reason, it may be important to select database page sizes that will be written as single block transfers by the underlying operating system. If you do not select a page size that the underlying operating system will write as a single block, you may want to configure the database to use checksums (see the DB->set_flags() flag for more information). By configuring checksums, you guarantee this kind of corruption will be detected at the expense of the CPU required to generate the checksums. When such an error is detected, the only course of recovery is to perform catastrophic recovery to restore the database.

Second, if you are copying database files (either as part of doing a hot backup or creation of a hot failover area), there is an additional question related to the page size of the Berkeley DB databases. You must copy databases atomically, in units of the database page size. In other words, the reads made by the copy program must not be interleaved with writes by other threads of control, and the copy program must read the databases in multiples of the underlying database page size. Generally, this is not a problem, as operating systems already make this guarantee and system utilities normally read in power-of-2 sized chunks, which are larger than the largest possible Berkeley DB database page size.

One problem we have seen in this area was in some releases of Solaris where the cp utility was implemented using the mmap system call rather than the read system call. Because the Solaris' mmap system call did not make the same guarantee of read atomicity as the read system call, using the cp utility could create corrupted copies of the databases. Another problem we have seen is implementations of the tar utility doing 10KB block reads by default, and even when an output block size was specified to that utility, not reading from the underlying databases in multiples of the block size. Using the dd utility instead of the cp or tar utilities (and specifying an appropriate block size), fixes these problems. If you plan to use a system utility to copy database files, you may want to use a system call trace utility (for example, ktrace or truss) to check for an I/O size smaller than or not a multiple of the database page size and system calls other than read.

Third, it is necessary to consider the behavior of the system's underlying stable storage hardware. For example, consider a SCSI controller that has been configured to cache data and return to the operating system that the data has been written to stable storage, when, in fact, it has only been written into the controller RAM cache. If power is lost before the controller is able to flush its cache to disk, and the controller cache is not stable (that is, the writes will not be flushed to disk when power returns), the writes will be lost. If the writes include database blocks, there is no loss because recovery will correctly update the database. If the writes include log file blocks, it is possible that transactions that were already committed may not appear in the recovered database, although the recovered database will be coherent after a crash.

If the underlying hardware can fail in any way so that only part of the block was written, the failure conditions are the same as those described previously for an operating system failure

that writes only part of a logical database block. In such cases, configuring the database for checksums will ensure the corruption is detected.

For these reasons, it may be important to select hardware that does not do partial writes and does not cache data writes (or does not return that the data has been written to stable storage until it has either been written to stable storage or the actual writing of all of the data is guaranteed, barring catastrophic hardware failure -- that is, your disk drive exploding).

If the disk drive on which you are storing your databases explodes, you can perform normal Berkeley DB catastrophic recovery, because it requires only a snapshot of your databases plus the log files you have archived since those snapshots were taken. In this case, you should lose no database changes at all.

If the disk drive on which you are storing your log files explodes, you can also perform catastrophic recovery, but you will lose any database changes made as part of transactions committed since your last archival of the log files. Alternatively, if your database environment and databases are still available after you lose the log file disk, you should be able to dump your databases. However, you may see an inconsistent snapshot of your data after doing the dump, because changes that were part of transactions that were not yet committed may appear in the database dump. Depending on the value of the data, a reasonable alternative may be to perform both the database dump and the catastrophic recovery and then compare the databases created by the two methods.

Regardless, for these reasons, storing your databases and log files on different disks should be considered a safety measure as well as a performance enhancement.

Finally, you should be aware that Berkeley DB does not protect against all cases of stable storage hardware failure, nor does it protect against simple hardware misbehavior (for example, a disk controller writing incorrect data to the disk). However, configuring the database for checksums will ensure that any such corruption is detected.

Transaction tuning

There are a few different issues to consider when tuning the performance of Berkeley DB transactional applications. First, you should review Access method tuning (page 79), as the tuning issues for access method applications are applicable to transactional applications as well. The following are additional tuning issues for Berkeley DB transactional applications:

access method

Highly concurrent applications should use the Queue access method, where possible, as it provides finer-granularity of locking than the other access methods. Otherwise, applications usually see better concurrency when using the Btree access method than when using either the Hash or Recno access methods.

record numbers

Using record numbers outside of the Queue access method will often slow down concurrent applications as they limit the degree of concurrency available in the database. Using the Recno access method, or the Btree access method with retrieval by record number configured can slow applications down.

Btree database size

When using the Btree access method, applications supporting concurrent access may see excessive numbers of deadlocks in small databases. There are two different approaches to resolving this problem. First, as the Btree access method uses page-level locking, decreasing the database page size can result in fewer lock conflicts. Second, in the case of databases that are cyclically growing and shrinking, turning off reverse splits (with DB_REVSPLITOFF) can leave the database with enough pages that there will be fewer lock conflicts.

read locks

Performing all read operations outside of transactions or at Degrees of isolation (page 163) can often significantly increase application throughput. In addition, limiting the lifetime of non-transactional cursors will reduce the length of times locks are held, thereby improving concurrency.

DB_DIRECT_DB, DB_LOG_DIRECT

On some systems, avoiding caching in the operating system can improve write throughput and allow the creation of larger Berkeley DB caches.

DB_READ_UNCOMMITTED, DB_READ_COMMITTED

Consider decreasing the level of isolation of transaction using the DB_READ_UNCOMMITTED, or DB_READ_COMMITTED flags for transactions or cursors or the DB_READ_UNCOMMITTED flag on individual read operations. The DB_READ_COMMITTED flag will release read locks on cursors as soon as the data page is nolonger referenced. This is also called *degree 2 isolation*. This will tend to block write operations for shorter periods for applications that do not need to have repeatable reads for cursor operations.

The DB_READ_COMMITTED flag will allow read operations to potentially return data which has been modified but not yet committed, and can significantly increase application throughput in applications that do not require data be guaranteed to be permanent in the database. This is also called *degree 1 isolation*, or *dirty reads*.

DB_RMW

If there are many deadlocks, consider using the DB_RMW flag to immediately acquire write locks when reading data items that will subsequently be modified. Although this flag may increase contention (because write locks are held longer than they would otherwise be), it may decrease the number of deadlocks that occur.

DB_TXN_WRITE_NOSYNC, DB_TXN_NOSYNC

By default, transactional commit in Berkeley DB implies durability, that is, all committed operations will be present in the database after recovery from any application or system failure. For applications not requiring that level of certainty, specifying the DB_TXN_NOSYNC flag will often provide a significant performance improvement. In this case, the database will still be fully recoverable, but some number of committed transactions might be lost after application or system failure.

access databases in order

When modifying multiple databases in a single transaction, always access physical files and databases within physical files, in the same order where possible. In addition, avoid returning to a physical file or database, that is, avoid accessing a database,

moving on to another database and then returning to the first database. This can significantly reduce the chance of deadlock between threads of control.

large key/data items

Transactional protections in Berkeley DB are guaranteed by before and after physical image logging. This means applications modifying large key/data items also write large log records, and, in the case of the default transaction commit, threads of control must wait until those log records have been flushed to disk. Applications supporting concurrent access should try and keep key/data items small wherever possible.

mutex selection

During configuration, Berkeley DB selects a mutex implementation for the architecture. Berkeley DB normally prefers blocking-mutex implementations over non-blocking ones. For example, Berkeley DB will select POSIX pthread mutex interfaces rather than assembly-code test-and-set spin mutexes because pthread mutexes are usually more efficient and less likely to waste CPU cycles spinning without getting any work accomplished.

For some applications and systems (generally highly concurrent applications on large multiprocessor systems), Berkeley DB makes the wrong choice. In some cases, better performance can be achieved by configuring with the --with-mutex argument and selecting a different mutex implementation than the one selected by Berkeley DB. When a test-and-set spin mutex implementation is selected, it may be useful to tune the number of spins made before yielding the processor and sleeping. For more information, see the DB_ENV->mutex_set_tas_spins() method.

Finally, Berkeley DB may put multiple mutexes on individual cache lines. When tuning Berkeley DB for large multiprocessor systems, it may be useful to tune mutex alignment using the DB_ENV->mutex_set_align() method.

--enable-posixmutexes

By default, the Berkeley DB library will only select the POSIX pthread mutex implementation if it supports mutexes shared between multiple processes. If your application does not share its database environment between processes and your system's POSIX mutex support was not selected because it did not support inter-process mutexes, you may be able to increase performance and transactional throughput by configuring with the --enable-posixmutexes argument.

log buffer size

Berkeley DB internally maintains a buffer of log writes. The buffer is written to disk at transaction commit, by default, or, whenever it is filled. If it is consistently being filled before transaction commit, it will be written multiple times per transaction, costing application performance. In these cases, increasing the size of the log buffer can increase application throughput.

log file location

If the database environment's log files are on the same disk as the databases, the disk arms will have to seek back-and-forth between the two. Placing the log files and the databases on different disk arms can often increase application throughput.

trickle write

In some applications, the cache is sufficiently active and dirty that readers frequently need to write a dirty page in order to have space in which to read a new page from the backing database file. You can use the db_stat utility (or the statistics returned by the DB_ENV->memp_stat() method) to see how often this is happening in your application's cache. In this case, using a separate thread of control and the DB_ENV->memp_trickle() method to trickle-write pages can often increase the overall throughput of the application.

Transaction throughput

Generally, the speed of a database system is measured by the *transaction throughput*, expressed as a number of transactions per second. The two gating factors for Berkeley DB performance in a transactional system are usually the underlying database files and the log file. Both are factors because they require disk I/O, which is slow relative to other system resources such as CPU.

In the worst-case scenario:

- Database access is truly random and the database is too large for any significant percentage of it to fit into the cache, resulting in a single I/O per requested key/data pair.
- Both the database and the log are on a single disk.

This means that for each transaction, Berkeley DB is potentially performing several filesystem operations:

- Disk seek to database file
- Database file read
- Disk seek to log file
- Log file write
- Flush log file information to disk
- Disk seek to update log file metadata (for example, inode information)
- Log metadata write
- Flush log file metadata to disk

There are a number of ways to increase transactional throughput, all of which attempt to decrease the number of filesystem operations per transaction. First, the Berkeley DB software includes support for *group commit*. Group commit simply means that when the information about one transaction is flushed to disk, the information for any other waiting transactions will be flushed to disk at the same time, potentially amortizing a single log write over a large number of transactions. There are additional tuning parameters which may be useful to application writers:

- Tune the size of the database cache. If the Berkeley DB key/data pairs used during the transaction are found in the database cache, the seek and read from the database are no longer necessary, resulting in two fewer filesystem operations per transaction. To determine whether your cache size is too small, see Selecting a cache size (page 20).

- Put the database and the log files on different disks. This allows reads and writes to the log files and the database files to be performed concurrently.

- Set the filesystem configuration so that file access and modification times are not updated. Note that although the file access and modification times are not used by Berkeley DB, this may affect other programs -- so be careful.

- Upgrade your hardware. When considering the hardware on which to run your application, however, it is important to consider the entire system. The controller and bus can have as much to do with the disk performance as the disk itself. It is also important to remember that throughput is rarely the limiting factor, and that disk seek times are normally the true performance issue for Berkeley DB.

- Turn on the DB_TXN_NOSYNC or DB_TXN_NOSYNCflags. This changes the Berkeley DB behavior so that the log files are not written and/or flushed when transactions are committed. Although this change will greatly increase your transaction throughput, it means that transactions will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability). Database integrity will be maintained, but it is possible that some number of the most recently committed transactions may be undone during recovery instead of being redone.

If you are bottlenecked on logging, the following test will help you confirm that the number of transactions per second that your application does is reasonable for the hardware on which you're running. Your test program should repeatedly perform the following operations:

- Seek to the beginning of a file

- Write to the file

- Flush the file write to disk

The number of times that you can perform these three operations per second is a rough measure of the minimum number of transactions per second of which the hardware is capable. This test simulates the operations applied to the log file. (As a simplifying assumption in this experiment, we assume that the database files are either on a separate disk; or that they fit, with some few exceptions, into the database cache.) We do not have to directly simulate updating the log file directory information because it will normally be updated and flushed to disk as a result of flushing the log file write to disk.

Running this test program, in which we write 256 bytes for 1000 operations on reasonably standard commodity hardware (Pentium II CPU, SCSI disk), returned the following results:

```
% testfile -b256 -o1000
running: 1000 ops
Elapsed time: 16.641934 seconds
1000 ops:   60.09 ops per second
```

Note that the number of bytes being written to the log as part of each transaction can dramatically affect the transaction throughput. The test run used 256, which is a reasonable size log write. Your log writes may be different. To determine your average log write size, use the db_stat utility to display your log statistics.

As a quick sanity check, the average seek time is 9.4 msec for this particular disk, and the average latency is 4.17 msec. That results in a minimum requirement for a data transfer to the disk of 13.57 msec, or a maximum of 74 transfers per second. This is close enough to the previous 60 operations per second (which wasn't done on a quiescent disk) that the number is believable.

An implementation of the previous example test program [writetest.cs] for IEEE/ANSI Std 1003.1 (POSIX) standard systems is included in the Berkeley DB distribution.

Transaction FAQ

1. **What should a transactional program do when an error occurs?**

   Any time an error occurs, such that a transactionally protected set of operations cannot complete successfully, the transaction must be aborted. While deadlock is by far the most common of these errors, there are other possibilities; for example, running out of disk space for the filesystem. In Berkeley DB transactional applications, there are three classes of error returns: "expected" errors, "unexpected but recoverable" errors, and a single "unrecoverable" error. Expected errors are errors like DB_NOTFOUND (page 230), which indicates that a searched-for key item is not present in the database. Applications may want to explicitly test for and handle this error, or, in the case where the absence of a key implies the enclosing transaction should fail, simply call DB_TXN->abort(). Unexpected but recoverable errors are errors like DB_LOCK_DEADLOCK (page 230), which indicates that an operation has been selected to resolve a deadlock, or a system error such as EIO, which likely indicates that the filesystem has no available disk space. Applications must immediately call DB_TXN->abort() when these returns occur, as it is not possible to proceed otherwise. The only unrecoverable error is DB_RUNRECOVERY (page 230), which indicates that the system must stop and recovery must be run.

2. **How can hot backups work? Can't you get an inconsistent picture of the database when you copy it?**

   First, Berkeley DB is based on the technique of "write-ahead logging", which means that before any change is made to a database, a log record is written that describes the change. Further, Berkeley DB guarantees that the log record that describes the change will always be written to stable storage (that is, disk) before the database page where the change was made is written to stable storage. Because of this guarantee, we know that any change made to a database will appear either in just a log file, or both the database and a log file, but never in just the database.

   Second, you can always create a consistent and correct database based on the log files and the databases from a database environment. So, during a hot backup, we first make a copy of the databases and then a copy of the log files. The tricky part is that there may be pages in the database that are related for which we won't get a consistent picture during this copy. For example, let's say that we copy pages 1-4 of the database, and then are swapped out.

For whatever reason (perhaps because we needed to flush pages from the cache, or because of a checkpoint), the database pages 1 and 5 are written. Then, the hot backup process is re-scheduled, and it copies page 5. Obviously, we have an inconsistent database snapshot, because we have a copy of page 1 from before it was written by the other thread of control, and a copy of page 5 after it was written by the other thread. What makes this work is the order of operations in a hot backup. Because of the write-ahead logging guarantees, we know that any page written to the database will first be referenced in the log. If we copy the database first, then we can also know that any inconsistency in the database will be described in the log files, and so we know that we can fix everything up during recovery.

3. **My application has DB_LOCK_DEADLOCK (page 230) errors. Is the normal, and what should I do?**

   It is quite rare for a transactional application to be deadlock free. All applications should be prepared to handle deadlock returns, because even if the application is deadlock free when deployed, future changes to the application or the Berkeley DB implementation might introduce deadlocks.

   Practices which reduce the chance of deadlock include:

   - Not using cursors which move backwards through the database (DB_PREV), as backward scanning cursors can deadlock with page splits;
   - Configuring DB_REVSPLITOFF to turn off reverse splits in applications which repeatedly delete and re-insert the same keys, to minimize the number of page splits as keys are re-inserted;
   - Not configuring DB_READ_UNCOMMITTED as that flag requires write transactions upgrade their locks when aborted, which can lead to deadlock. Generally, DB_READ_COMMITTED or non-transactional read operations are less prone to deadlock than DB_READ_UNCOMMITTED.

4. **How can I move a database from one transactional environment into another?**

   Because database pages contain references to log records, databases cannot be simply moved into different database environments. To move a database into a different environment, dump and reload the database before moving it. If the database is too large to dump and reload, the database may be prepared in place using the DB_ENV->lsn_reset() method or the **-r** argument to the db_load utility.

5. **I'm seeing the error "log_flush: LSN past current end-of-log", what does that mean?**

   The most common cause of this error is that a system administrator has removed all of the log files from a database environment. You should shut down your database environment as gracefully as possible, first flushing the database environment cache to disk, if that's possible. Then, dump and reload your databases. If the database is too large to dump and reload, the database may be reset in place using the DB_ENV->lsn_reset() method or the **-r** argument to the db_load utility. However, if you reset the database in place, you should verify your databases before using them again. (It is possible for the databases to be corrupted by

running after all of the log files have been removed, and the longer the application runs, the worse it can get.)

Chapter 12. Berkeley DB Replication

Replication introduction

Berkeley DB includes support for building highly available applications based on replication. Berkeley DB replication groups consist of some number of independently configured database environments. There is a single *master* database environment and one or more *client* database environments. Master environments support both database reads and writes; client environments support only database reads. If the master environment fails, applications may upgrade a client to be the new master. The database environments might be on separate computers, on separate hardware partitions in a non-uniform memory access (NUMA) system, or on separate disks in a single server. As always with Berkeley DB environments, any number of concurrent processes or threads may access a database environment. In the case of a master environment, any number of threads of control may read and write the environment, and in the case of a client environment, any number of threads of control may read the environment.

Applications may be written to provide various degrees of consistency between the master and clients. The system can be run synchronously such that replicas are guaranteed to be up-to-date with all committed transactions, but doing so may incur a significant performance penalty. Higher performance solutions sacrifice total consistency, allowing the clients to be out of date for an application-controlled amount of time.

There are two ways to build replicated applications. The simpler way is to use the Berkeley DB Replication Manager. The Replication Manager provides a standard communications infrastructure, and it creates and manages the background threads needed for processing replication messages. (Note that in Replication Manager applications, all updates to databases at the master environment must be done through a single DB_ENV environment handle, though they may occur in multiple threads. This of course means that only a single process may update data.)

The Replication Manager implementation is based on TCP/IP sockets, and uses POSIX 1003.1 style networking and thread support. (On Windows systems, it uses standard Windows thread support.) As a result, it is not as portable as the rest of the Berkeley DB library itself.

The alternative is to use the lower-level replication "Base APIs". This approach affords more flexibility, but requires the application to provide some critical components:

1. A communication infrastructure. Applications may use whatever wire protocol is appropriate for their application (for example, RPC, TCP/IP, UDP, VI or message-passing over the backplane).

2. The application is responsible for naming. Berkeley DB refers to the members of a replication group using an application-provided ID, and applications must map that ID to a particular database environment or communication channel.

3. The application is responsible for monitoring the status of the master and clients, and identifying any unavailable database environments.

4. The application must provide whatever security policies are needed. For example, the application may choose to encrypt data, use a secure socket layer, or do nothing at all. The level of security is left to the sole discretion of the application.

(Note that Replication Manager does not provide wire security for replication messages.)

The following pages present various programming considerations, many of which are directly relevant only for Base API applications. However, even when using Replication Manager it is important to understand the concepts.

Finally, the Berkeley DB replication implementation has one other additional feature to increase application reliability. Replication in Berkeley DB is implemented to perform database updates using a different code path than the standard ones. This means operations that manage to crash the replication master due to a software bug will not necessarily also crash replication clients.

Replication Manager Methods	Description
DB_ENV->repmgr_add_remote_site()	Specify the Replication Manager's remote sites
DB_ENV->repmgr_set_ack_policy()	Specify the Replication Manager's client acknowledgement policy
DB_ENV->repmgr_set_local_site()	Specify the Replication Manager's local site
DB_ENV->repmgr_site_list()	List the sites and their status
DB_ENV->repmgr_start()	Start the Replication Manager
DB_ENV->repmgr_stat()	Replication Manager statistics
DB_ENV->repmgr_stat_print()	Print Replication Manager statistics
Base API Methods	
DB_ENV->rep_elect()	Hold a replication election
DB_ENV->rep_process_message()	Process a replication message
DB_ENV->rep_set_transport()	Configure replication transport callback
DB_ENV->rep_start()	Start replication
Additional Replication Methods	
DB_ENV->rep_stat()	Replication statistics
DB_ENV->rep_stat_print()	Print replication statistics
DB_ENV->rep_sync()	Replication synchronization
Replication Configuration	
DB_ENV->rep_set_clockskew()	Configure master lease clock adjustment
DB_ENV->rep_set_config()	Configure the replication subsystem
DB_ENV->rep_set_limit()	Limit data sent in response to a single message
DB_ENV->rep_set_nsites()	Configure replication group site count
DB_ENV->rep_set_priority()	Configure replication site priority

Replication Manager Methods	Description
DB_ENV->rep_set_request()	Configure replication client retransmission requests
DB_ENV->rep_set_timeout()	Configure replication timeouts

Replication environment IDs

Each database environment included in a replication group must have a unique identifier for itself and for the other members of the replication group. The identifiers do not need to be global, that is, each database environment can assign local identifiers to members of the replication group as it encounters them. For example, given three sites: A, B and C, site A might assign the identifiers 1 and 2 to sites B and C respectively, while site B might assign the identifiers 301 and 302 to sites A and C respectively. Note that it is not wrong to have global identifiers, it is just not a requirement.

Replication Manager assigns and manages environment IDs on behalf of the application.

It is the responsibility of a Base API application to label each incoming replication message passed to DB_ENV->rep_process_message() method with the appropriate identifier. Subsequently, Berkeley DB will label outgoing messages to the **send** function with those same identifiers.

Negative identifiers are reserved for use by Berkeley DB, and should never be assigned to environments by the application. Two of these reserved identifiers are intended for application use, as follows:

DB_EID_BROADCAST

The DB_EID_BROADCAST identifier indicates a message should be broadcast to all members of a replication group.

DB_EID_INVALID

The DB_EID_INVALID identifier is an invalid environment ID, and may be used to initialize environment ID variables that are subsequently checked for validity.

Replication environment priorities

Each database environment included in a replication group must have a priority, which specifies a relative ordering among the different environments in a replication group. This ordering is a factor in determining which environment will be selected as a new master in case the existing master fails. Both Replication Manager applications and Base API applications should specify environment priorities.

Priorities are an unsigned integer, but do not need to be unique throughout the replication group. A priority of 0 means the system can never become a master. Otherwise, larger valued priorities indicate a more desirable master. For example, if a replication group consists of three database environments, two of which are connected by an OC3 and the third of which is connected by a T1, the third database environment should be assigned a priority value which is lower than either of the other two.

Desirability of the master is first determined by the client having the most recent log records. Ties in log records are broken with the client priority. If both sites have the same number of log records and the same priority, one is selected at random.

Building replicated applications

The simplest way to build a replicated Berkeley DB application is to first build (and debug!) the transactional version of the same application. Then, add a thin replication layer: application initialization must be changed and the application's communication infrastructure must be added.

The application initialization changes are relatively simple. Replication Manager provides a communication infrastructure, but in order to use the replication Base APIs you must provide your own.

For implementation reasons, all replicated databases must reside in the data directories set from DB_ENV->set_data_dir() (or in the default environment home directory, if not using DB_ENV->set_data_dir()), rather than in a subdirectory below the specified directory. Care must be taken in applications using relative pathnames and changing working directories after opening the environment. In such applications the replication initialization code may not be able to locate the databases, and applications that change their working directories may need to use absolute pathnames.

During application initialization, the application performs three additional tasks: first, it must specify the DB_INIT_REP flag when opening its database environment and additionally, a Replication Manager application must also specify the DB_THREAD flag; second, it must provide Berkeley DB information about its communications infrastructure; and third, it must start the Berkeley DB replication system. Generally, a replicated application will do normal Berkeley DB recovery and configuration, exactly like any other transactional application.

Replication Manager applications configure the built-in communications infrastructure by calling the DB_ENV->repmgr_set_local_site() method once and the DB_ENV->repmgr_add_remote_site() method zero or more times. Once the environment has been opened, the application starts the replication system by calling the DB_ENV->repmgr_start() method.

A Base API application calls the DB_ENV->rep_set_transport() method to configure the entry point to its own communications infrastructure, and then calls the DB_ENV->rep_start() method to join or create the replication group.

When starting the replication system, an application has two choices: it may choose the group master site explicitly, or alternatively it may configure all group members as clients and then call for an election, letting the clients select the master from among themselves. Either is correct, and the choice is entirely up to the application.

Replication Manager applications make this choice simply by setting the flags parameter to the DB_ENV->repmgr_start() method.

For a Base API application, the result of calling DB_ENV->rep_start() is usually the discovery of a master, or the declaration of the local environment as the master. If a master has not

been discovered after a reasonable amount of time, the application should call DB_ENV->rep_elect() to call for an election.

Consider a Base API application with multiple processes or multiple environment handles that modify databases in the replicated environment. All modifications must be done on the master environment. The first process to join or create the master environment must call both the DB_ENV->rep_set_transport() and the DB_ENV->rep_start() method. Subsequent replication processes must at least call the DB_ENV->rep_set_transport() method. Those processes may call the DB_ENV->rep_start() method (as long as they use the same master or client argument). If multiple processes are modifying the master environment there must be a unified communication infrastructure such that messages arriving at clients have a single master ID. Additionally the application must be structured so that all incoming messages are able to be processed by a single DB_ENV handle.

Note that not all processes running in replicated environments need to call DB_ENV->repmgr_start(), DB_ENV->rep_set_transport() or DB_ENV->rep_start(). Read-only processes running in a master environment do not need to be configured for replication in any way. Processes running in a client environment are read-only by definition, and so do not need to be configured for replication either (although, in the case of clients that may become masters, it is usually simplest to configure for replication on process startup rather than trying to reconfigure when the client becomes a master). Obviously, at least one thread of control on each client must be configured for replication as messages must be passed between the master and the client.

Any site in a replication group may have its own private transactional databases in the environment as well. A site may create a local database by specifying the DB_TXN_NOT_DURABLE flag to the DB->set_flags() method. The application must never create a private database with the same name as a database replicated across the entire environment as data corruption can result.

For implementation reasons, Base API applications must process all incoming replication messages using the same DB_ENV handle. It is not required that a single thread of control process all messages, only that all threads of control processing messages use the same handle.

No additional calls are required to shut down a database environment participating in a replication group. The application should shut down the environment in the usual manner, by calling the DB_ENV->close() method. For Replication Manager applications, this also terminates all network connections and background processing threads.

Replication Manager methods

Applications which use the Replication Manager support generally call the following Berkeley DB methods. The general pattern is to call various methods to configure Replication Manager, and then start it by calling DB_ENV->repmgr_start(). Once this initialization is complete, the application rarely needs to call any of these methods. (A prime example of an exception to this rule would be the DB_ENV->rep_sync() method, if the application is Delaying client synchronization (page 203).)

DB_ENV->repmgr_set_local_site()

The DB_ENV->repmgr_set_local_site() method configures the TCP/IP address of the local site, by specifying the port number on which it is to listen for incoming connection requests. This method must be called exactly once before calling DB_ENV->repmgr_start().

DB_ENV->repmgr_add_remote_site()

The DB_ENV->repmgr_add_remote_site() method adds a remote site to the list of sites initially known by the local site. The remote site is specified in terms of a TCP/IP network address: host name and port number. This method may be called as many times as necessary to configure all known remote sites. Note that it is usually not necessary for each site in the replication group initially to know about all other sites in the group. Sites can discover each other dynamically, as described in Connecting to a new site (page 198).

DB_ENV->repmgr_set_ack_policy()

The DB_ENV->repmgr_set_ack_policy() method configures the acknowledgement policy to be used in the replication group, in other words, the behavior of the master with respect to acknowledgements for "permanent" messages, which implements the application's requirements for Transactional guarantees (page 206). The current implementation requires all sites in the replication group to configure the same acknowledgement policy.

DB_ENV->rep_set_nsites()

The DB_ENV->rep_set_nsites() method tells Replication Manager the total number of sites in the replication group. This is usually necessary for proper operation of elections, and for counting message acknowledgements (depending on the acknowledgement policy).

DB_ENV->rep_set_priority()

The DB_ENV->rep_set_priority() method configures the local site's priority for the purpose of elections.

DB_ENV->rep_set_timeout()

This method optionally configures various timeout values. Otherwise default timeout values as specified in DB_ENV->rep_set_timeout() are used. In particular, Replication Manager client sites can be configured to monitor the health of the TCP/IP connection to the master site using heartbeat messages. If the client receives no messages from the master for a certain amount of time, it considers the connection to be broken, and calls for an election to choose a new master.

DB_ENV->set_event_notify()

Once configured and started, Replication Manager does virtually all of its work in the background, usually without the need for any direct communication with the application. However, occasionally events occur which the application may be interested in knowing about. The application can request notification of these events by calling the DB_ENV->set_event_notify() method.

DB_ENV->repmgr_start()

The DB_ENV->repmgr_start() method starts the replication system. It opens the listening TCP/IP socket and creates all the background processing threads that will be needed.

In addition to the methods previously described, Replication Manager applications may also call the following methods, as needed: DB_ENV->rep_set_config(), DB_ENV->rep_set_limit(), DB_ENV->rep_set_request(), DB_ENV->rep_sync() and DB_ENV->rep_stat().

Base API Methods

Base API applications use the following Berkeley DB methods.

DB_ENV->rep_set_transport()

The DB_ENV->rep_set_transport() method configures the replication system's communications infrastructure.

DB_ENV->rep_start()

The DB_ENV->rep_start() method configures (or reconfigures) an existing database environment to be a replication master or client.

DB_ENV->rep_process_message()

The DB_ENV->rep_process_message() method is used to process incoming messages from other environments in the replication group. For clients, it is responsible for accepting log records and updating the local databases based on messages from the master. For both the master and the clients, it is responsible for handling administrative functions (for example, the protocol for dealing with lost messages), and permitting new clients to join an active replication group. This method should only be called after the replication system's communications infrastructure has been configured via DB_ENV->rep_set_transport().

DB_ENV->rep_elect()

The DB_ENV->rep_elect() method causes the replication group to elect a new master; it is called whenever contact with the master is lost and the application wants the remaining sites to select a new master.

DB_ENV->set_event_notify()

The DB_ENV->set_event_notify() method is needed for applications to discover important replication-related events, such as the result of an election and appointment of a new master.

DB_ENV->rep_set_priority()

The DB_ENV->rep_set_priority() method configures the local site's priority for the purpose of elections.

DB_ENV->rep_set_timeout()

This method optionally configures various timeout values. Otherwise default timeout values as specified in DB_ENV->rep_set_timeout() are used.

DB_ENV->rep_set_limit()

The DB_ENV->rep_set_limit() method imposes an upper bound on the amount of data that will be sent in response to a single call to DB_ENV->rep_process_message(). During client recovery, that is, when a replica site is trying to synchronize with the master, clients may ask the master for a large number of log records. If it is going to harm an application for the master message loop to remain busy for an extended period transmitting records to the replica, then the application will want to use

DB_ENV->rep_set_limit() to limit the amount of data the master will send before relinquishing control and accepting other messages.

DB_ENV->rep_set_request()

This method sets a threshold for the minimum and maximum time that a client waits before requesting retransmission of a missing message.

In addition to the methods previously described, Base API applications may also call the following methods, as needed: DB_ENV->rep_stat(), DB_ENV->rep_sync() and DB_ENV->rep_set_config().

Building the communications infrastructure

Replication Manager provides a built-in communications infrastructure.

Base API applications must provide their own communications infrastructure, which is typically written with one or more threads of control looping on one or more communication channels, receiving and sending messages. These threads accept messages from remote environments for the local database environment, and accept messages from the local environment for remote environments. Messages from remote environments are passed to the local database environment using the DB_ENV->rep_process_message() method. Messages from the local environment are passed to the application for transmission using the callback function specified to the DB_ENV->rep_set_transport() method.

Processes establish communication channels by calling the DB_ENV->rep_set_transport() method, regardless of whether they are running in client or server environments. This method specifies the **send** function, a callback function used by Berkeley DB for sending messages to other database environments in the replication group. The **send** function takes an environment ID and two opaque data objects. It is the responsibility of the **send** function to transmit the information in the two data objects to the database environment corresponding to the ID, with the receiving application then calling the DB_ENV->rep_process_message() method to process the message.

The details of the transport mechanism are left entirely to the application; the only requirement is that the data buffer and size of each of the control and rec DBTs passed to the **send** function on the sending site be faithfully copied and delivered to the receiving site by means of a call to DB_ENV->rep_process_message() with corresponding arguments. Messages that are broadcast (whether by broadcast media or when directed by setting the DB_ENV->rep_set_transport() method's envid parameter DB_EID_BROADCAST), should not be processed by the message sender. In all cases, the application's transport media or software must ensure that DB_ENV->rep_process_message() is never called with a message intended for a different database environment or a broadcast message sent from the same environment on which DB_ENV->rep_process_message() will be called. The DB_ENV->rep_process_message() method is free-threaded; it is safe to deliver any number of messages simultaneously, and from any arbitrary thread or process in the Berkeley DB environment.

There are a number of informational returns from the DB_ENV->rep_process_message() method:

DB_REP_DUPMASTER

When DB_ENV->rep_process_message() returns DB_REP_DUPMASTER, it means that another database environment in the replication group also believes itself to be the

master. The application should complete all active transactions, close all open database handles, reconfigure itself as a client using the DB_ENV->rep_start() method, and then call for an election by calling the DB_ENV->rep_elect() method.

DB_REP_HOLDELECTION

When DB_ENV->rep_process_message() returns DB_REP_HOLDELECTION, it means that another database environment in the replication group has called for an election. The application should call the DB_ENV->rep_elect() method.

DB_REP_IGNORE

When DB_ENV->rep_process_message() returns DB_REP_IGNORE, it means that this message cannot be processed. This is normally an indication that this message is irrelevant to the current replication state, such as a message from an old generation that arrived late.

DB_REP_ISPERM

When DB_ENV->rep_process_message() returns DB_REP_ISPERM, it means a permanent record, perhaps a message previously returned as DB_REP_NOTPERM, was successfully written to disk. This record may have filled a gap in the log record that allowed additional records to be written. The **ret_lsnp** contains the maximum LSN of the permanent records written.

DB_REP_NEWSITE

When DB_ENV->rep_process_message() returns DB_REP_NEWSITE, it means that a message from a previously unknown member of the replication group has been received. The application should reconfigure itself as necessary so it is able to send messages to this site.

DB_REP_NOTPERM

When DB_ENV->rep_process_message() returns DB_REP_NOTPERM, it means a message marked as DB_REP_PERMANENT was processed successfully but was not written to disk. This is normally an indication that one or more messages, which should have arrived before this message, have not yet arrived. This operation will be written to disk when the missing messages arrive. The **ret_lsnp** argument will contain the LSN of this record. The application should take whatever action is deemed necessary to retain its recoverability characteristics.

Connecting to a new site

To add a new site to the replication group all that is needed is for the client member to join. Berkeley DB will perform an internal initialization from the master to the client automatically and will run recovery on the client to bring it up to date with the master.

For Base API applications, connecting to a new site in the replication group happens whenever the DB_ENV->rep_process_message() method returns DB_REP_NEWSITE. The application should assign the new site a local environment ID number, and all future messages from the site passed to DB_ENV->rep_process_message() should include that environment ID number. It is possible, of course, for the application to be aware of a new site before the return of DB_ENV->rep_process_message() (for example, applications using connection-oriented protocols

are likely to detect new sites immediately, while applications using broadcast protocols may not).

Regardless, in applications supporting the dynamic addition of database environments to replication groups, environments joining an existing replication group may need to provide contact information. (For example, in an application using TCP/IP sockets, a DNS name or IP address might be a reasonable value to provide.) This can be done using the **cdata** parameter to the DB_ENV->rep_start() method. The information referenced by **cdata** is wrapped in the initial contact message sent by the new environment, and is provided to the existing members of the group using the **rec** parameter returned by DB_ENV->rep_process_message(). If no additional information was provided for Berkeley DB to forward to the existing members of the group, the **data** field of the **rec** parameter passed to the DB_ENV->rep_process_message() method will be NULL after DB_ENV->rep_process_message() returns DB_REP_NEWSITE.

Replication Manager automatically distributes contact information using the mechanisms previously described.

Running Replication Manager in multiple processes

Replication Manager supports shared access to a database environment from multiple processes.

One replication process and multiple subordinate processes

Each site in a replication group has just one network address (TCP/IP host name and port number). This means that only one process can accept incoming connections. At least one application process must invoke the DB_ENV->repmgr_start() method to initiate communications and management of the replication state.

If it is convenient, multiple processes may issue calls to the Replication Manager configuration methods, and multiple processes may call DB_ENV->repmgr_start(). Replication Manager automatically opens the TCP/IP listening socket in the first process to do so (we'll call it the "replication process" here), and ignores this step in any subsequent processes ("subordinate processes").

Persistence of network address configuration

Local and remote site network addresses are stored in shared memory, and remain intact even when (all) processes close their environment handles gracefully and terminate. A process which opens an environment handle without running recovery automatically inherits all existing network address configuration. Such a process may not change the local site address (although it is allowed to make a redundant call to repmgr_set_local_site() specifying a configuration matching that which is already in effect).

Similarly, removing an existing remote site address from an intact environment is currently not supported. In order to make either of these types of change, the application must run recovery. By doing so you start fresh with a clean slate, erasing all network address configuration information.

Programming considerations

Note that Replication Manager applications must follow all the usual rules for Berkeley DB multi-threaded and/or multi-process applications, such as ensuring that the recovery operation occurs single-threaded, only once, before any other thread or processes operate in the environment. Since Replication Manager creates its own background threads which operate on the environment, all environment handles must be opened with the DB_THREAD flag, even if the application is otherwise single-threaded per process.

At the replication master site, each Replication Manager process opens outgoing TCP/IP connections to all clients in the replication group. It uses these direct connections to send to clients any log records resulting from update transactions that the process executes. But all other replication activity —message processing, elections, etc.— takes place only in the replication process.

Replication Manager notifies the application of certain events, using the callback function configured with the DB_ENV->set_event_notify() method. These notifications occur only in the process where the event itself occurred. Generally this means that most notifications occur only in the "replication process". Currently the only replication notification that can occur in a "subordinate process" is DB_EVENT_REP_PERM_FAILED.

Handling failure

Multi-process Replication Manager applications should handle failures in a manner consistent with the rules described in Handling failure in Transactional Data Store applications (page 145). To summarize, there are two ways to handle failure of a process:

1. The simple way is to kill all remaining processes, run recovery, and then restart all processes from the beginning. But this can be a bit drastic.

2. Using the DB_ENV->failchk() method, it is sometimes possible to leave surviving processes running, and just restart the failed process.

   Multi-process Replication Manager applications using this technique must start a new process when an old process fails. It is not possible for a "subordinate process" to take over the duties of a failed "replication process". If the failed process happens to be the replication process, then after a failchk() call the next process to call DB_ENV->repmgr_start() will become the new replication process.

Other miscellaneous rules

1. A database environment may not be shared between a Replication Manager application process and a Base API application process.

2. It is not possible to run multiple Replication Manager processes during mixed-version live upgrades from Berkeley DB versions prior to 4.8.

Elections

Replication Manager automatically conducts elections when necessary, based on configuration information supplied to the DB_ENV->rep_set_priority() method.

It is the responsibility of a Base API application to initiate elections if desired. It is never dangerous to hold an election, as the Berkeley DB election process ensures there is never more than a single master database environment. Clients should initiate an election whenever they lose contact with the master environment, whenever they see a return of DB_REP_HOLDELECTION from the DB_ENV->rep_process_message() method, or when, for whatever reason, they do not know who the master is. It is not necessary for applications to immediately hold elections when they start, as any existing master will be discovered after calling DB_ENV->rep_start(). If no master has been found after a short wait period, then the application should call for an election.

For a client to win an election, the replication group must currently have no master, and the client must have the most recent log records. In the case of clients having equivalent log records, the priority of the database environments participating in the election will determine the winner. The application specifies the minimum number of replication group members that must participate in an election for a winner to be declared. We recommend at least ((N/2) + 1) members. If fewer than the simple majority are specified, a warning will be given.

If an application's policy for what site should win an election can be parameterized in terms of the database environment's information (that is, the number of sites, available log records and a relative priority are all that matter), then Berkeley DB can handle all elections transparently. However, there are cases where the application has more complete knowledge and needs to affect the outcome of elections. For example, applications may choose to handle master selection, explicitly designating master and client sites. Applications in these cases may never need to call for an election. Alternatively, applications may choose to use DB_ENV->rep_elect()'s arguments to force the correct outcome to an election. That is, if an application has three sites, A, B, and C, and after a failure of C determines that A must become the winner, the application can guarantee an election's outcome by specifying priorities appropriately after an election:

```
on A: priority 100, nsites 2
on B: priority 0, nsites 2
```

It is dangerous to configure more than one master environment using the DB_ENV->rep_start() method, and applications should be careful not to do so. Applications should only configure themselves as the master environment if they are the only possible master, or if they have won an election. An application knows it has won an election when it receives the DB_EVENT_REP_ELECTED event.

Normally, when a master failure is detected it is desired that an election finish quickly so the application can continue to service updates. Also, participating sites are already up and can participate. However, in the case of restarting a whole group after an administrative shutdown, it is possible that a slower booting site had later logs than any other site. To cover that case, an application would like to give the election more time to ensure all sites have a chance to participate. Since it is intractable for a starting site to determine which case the whole group is in, the use of a long timeout gives all sites a reasonable chance to participate. If an application wanting full participation sets the **nvotes** arg to the DB_ENV->rep_elect() method to the number

of sites in the group and one site does not reboot, a master can never be elected without manual intervention.

In those cases, the desired action at a group level is to hold a full election if all sites crashed and a majority election if a subset of sites crashed or rebooted. Since an individual site cannot know which number of votes to require, a mechanism is available to accomplish this using timeouts. By setting a long timeout (perhaps on the order of minutes) using the **DB_REP_FULL_ELECTION_TIMEOUT** flag to the DB_ENV->rep_set_timeout() method, an application can allow Berkeley DB to elect a master even without full participation. Sites may also want to set a normal election timeout for majority based elections using the **DB_REP_ELECTION_TIMEOUT** flag to the DB_ENV->rep_set_timeout() method.

Consider 3 sites, A, B, and C where A is the master. In the case where all three sites crash and all reboot, all sites will set a timeout for a full election, say 10 minutes, but only require a majority for **nvotes** to the DB_ENV->rep_elect() method. Once all three sites are booted the election will complete immediately if they reboot within 10 minutes of each other. Consider if all three sites crash and only two reboot. The two sites will enter the election, but after the 10 minute timeout they will elect with the majority of two sites. Using the full election timeout sets a threshold for allowing a site to reboot and rejoin the group.

To add a database environment to the replication group with the intent of it becoming the master, first add it as a client. Since it may be out-of-date with respect to the current master, allow it to update itself from the current master. Then, shut the current master down. Presumably, the added client will win the subsequent election. If the client does not win the election, it is likely that it was not given sufficient time to update itself with respect to the current master.

If a client is unable to find a master or win an election, it means that the network has been partitioned and there are not enough environments participating in the election for one of the participants to win. In this case, the application should repeatedly call DB_ENV->rep_start() and DB_ENV->rep_elect(), alternating between attempting to discover an existing master, and holding an election to declare a new one. In desperate circumstances, an application could simply declare itself the master by calling DB_ENV->rep_start(), or by reducing the number of participants required to win an election until the election is won. Neither of these solutions is recommended: in the case of a network partition, either of these choices can result in there being two masters in one replication group, and the databases in the environment might irretrievably diverge as they are modified in different ways by the masters.

Note that this presents a special problem for a replication group consisting of only two environments. If a master site fails, the remaining client can never comprise a majority of sites in the group. If the client application can reach a remote network site, or some other external tie-breaker, it may be able to determine whether it is safe to declare itself master. Otherwise it must choose between providing availability of a writable master (at the risk of duplicate masters), or strict protection against duplicate masters (but no master when a failure occurs). Replication Manager offers this choice via the DB_ENV->rep_set_config() method. Base API applications can accomplish this by judicious setting of the nvotes and nsites parameters to the DB_ENV->rep_elect() method.

It is possible for a less-preferred database environment to win an election if a number of systems crash at the same time. Because an election winner is declared as soon as enough

environments participate in the election, the environment on a slow booting but well-connected machine might lose to an environment on a badly connected but faster booting machine. In the case of a number of environments crashing at the same time (for example, a set of replicated servers in a single machine room), applications should bring the database environments on line as clients initially (which will allow them to process read queries immediately), and then hold an election after sufficient time has passed for the slower booting machines to catch up.

If, for any reason, a less-preferred database environment becomes the master, it is possible to switch masters in a replicated environment. For example, the preferred master crashes, and one of the replication group clients becomes the group master. In order to restore the preferred master to master status, take the following steps:

1. The preferred master should reboot and re-join the replication group as a client.

2. Once the preferred master has caught up with the replication group, the application on the current master should complete all active transactions and reconfigure itself as a client using the DB_ENV->rep_start() method.

3. Then, the current or preferred master should call for an election using the DB_ENV->rep_elect() method.

Synchronizing with a master

When a client detects a new replication group master, the client must synchronize with the new master before the client can process new database changes. Synchronizing is a heavyweight operation which can place a burden on both the client and the master. There are several controls an application can use to reduce the synchronization burden.

Delaying client synchronization

When a replication group has a new master, either as specified by the application or as a result of winning an election, all clients in the replication group must synchronize with the new master. This can strain the resources of the new master since a large number of clients may be attempting to communicate with and transfer records from the master. Client applications wanting to delay client synchronization should call the DB_ENV->rep_set_config() method with the DB_REP_CONF_DELAYCLIENT flag. The application will be notified of the establishment of the new master as usual, but the client will not proceed to synchronize with the new master.

Applications learn of a new master via the DB_EVENT_REP_NEWMASTER event.

Client applications choosing to delay synchronization in this manner are responsible for synchronizing the client environment at some future time using the DB_ENV->rep_sync() method.

Client-to-client synchronization

Instead of synchronizing with the new master, it is sometimes possible for a client to synchronize with another client. Berkeley DB initiates synchronization at the client by sending a request message via the transport call-back function of the communication infrastructure. The message is destined for the master site, but is also marked with a DB_REP_ANYWHERE flag. The

application may choose to send such a request to another client, or to ignore the flag, sending it to its indicated destination.

Furthermore, when the other client receives such a request it may be unable to satisfy it. In this case it will reply to the requesting client, telling it that it is unable to provide the requested information. The requesting client will then re-issue the request. Additionally, if the original request never reaches the other client, the requesting client will again re-issue the request. In either of these cases the message will be marked with the DB_REP_REREQUEST flag. The application may continue trying to find another client to service the request, or it may give up and simply send it to the master (that is, the environment ID explicitly specified to the transport function).

Replication Manager allows an application to designate one remote site (called its "peer") to receive client-to-client requests, via the flags parameter to the DB_ENV->repmgr_add_remote_site() method. Replication Manager will always first try to send requests marked with the DB_REP_ANYWHERE flag to its peer, if available. However, it will always send a DB_REP_REREQUEST to the master site.

Base API applications have complete freedom in choosing where to send these DB_REP_ANYWHERE requests, and in deciding how to handle DB_REP_REREQUEST.

The delayed synchronization and client-to-client synchronization features allow applications to do load balancing within replication groups. For example, consider a replication group with 5 sites, A, B, C, D and E. Site E just crashed, and site A was elected master. Sites C and D have been configured for delayed synchronization. When site B is notified that site A is a new master, it immediately synchronizes. When B finishes synchronizing with the master, the application calls the DB_ENV->rep_sync() method on sites C and D to cause them to synchronize as well. Sites C and D (and E, when it has finished rebooting) can send their requests to site B, and B then bears the brunt of the work and network traffic for synchronization, making master site A available to handle the normal application load and any write requests paused by the election.

Blocked client operations

Clients in the process of synchronizing with the master block access to Berkeley DB operations during some parts of that process. By default, most Berkeley DB methods will block until client synchronization is complete, and then the method call proceeds.

Client applications which cannot wait and would prefer an immediate error return instead of blocking, should call the DB_ENV->rep_set_config() method with the DB_REP_CONF_NOWAIT flag. This configuration causes DB method calls to immediately return a DB_REP_LOCKOUT error instead of blocking, if the client is currently synchronizing with the master.

Clients too far out-of-date to synchronize

Clients attempting to synchronize with the master may discover that synchronization is not possible because the client no longer has any overlapping information with the master site. By default, the master and client automatically detect this state and perform an internal initialization of the client. Because internal initialization requires transfer of entire databases to the client, it can take a relatively long period of time and may require database handles to be reopened in the client applications.

Client applications which cannot wait or would prefer to do a hot backup instead of performing internal initialization, should call the DB_ENV->rep_set_config() method with the DB_REP_CONF_NOAUTOINIT flag. This configuration flag causes Berkeley DB to return DB_REP_JOIN_FAILURE to the application instead of performing internal initialization.

Initializing a new site

By default, adding a new site to a replication group only requires the client to join. Berkeley DB will automatically perform internal initialization from the master to the client, bringing the client into sync with the master.

However, depending on the network and infrastructure, it can be advantageous in a few instances to use a "hot backup" to initialize a client into a replication group. Clients not wanting to automatically perform internal initialization should call the DB_ENV->rep_set_config() method with the DB_REP_CONF_NOAUTOINIT flag. This configuration flag causes Berkeley DB to return DB_REP_JOIN_FAILURE to the application's DB_ENV->rep_process_message() method instead of performing internal initialization.

To use a hot backup to initialize a client into a replication group, perform the following steps:

1. Do an archival backup of the master's environment, as described in Database and log file archival (page 173). The backup can either be a conventional backup or a hot backup.

2. Copy the archival backup into a clean environment directory on the client.

3. Run catastrophic recovery on the client's new environment, as described in Recovery procedures (page 176).

4. Reconfigure and reopen the environment as a client member of the replication group.

If copying the backup to the client takes a long time relative to the frequency with which log files are reclaimed using the db_archive utility or the DB_ENV->log_archive() method, it may be necessary to suppress log reclamation until the newly restarted client has "caught up" and applied all log records generated during its downtime.

As with any Berkeley DB application, the database environment must be in a consistent state at application startup. This is most easily assured by running recovery at startup time in one thread or process; it is harmless to do this on both clients and masters even when not strictly necessary.

Bulk transfer

Sites in a replication group may be configured to use bulk transfer by calling the DB_ENV->rep_set_config() method with the DB_REP_CONF_BULK flag. When configured for bulk transfer, sites will accumulate records in a buffer and transfer them to another site in a single network transfer. Configuring bulk transfer makes sense for master sites, of course. Additionally, applications using client-to-client synchronization may find it helpful to configure bulk transfer for client sites as well.

When a master is generating new log records, or any information request is made of a master, and bulk transfer has been configured, records will accumulate in a bulk buffer. The bulk buffer will be sent to the client if either the buffer is full or if a permanent record (for example, a transaction commit or checkpoint record) is queued for the client.

When a client is responding to another client's request for information, and bulk transfer has been configured, records will accumulate in a bulk buffer. The bulk buffer will be sent to the client when the buffer is full or when the client's request has been satisfied; no particular type of record will cause the buffer to be sent.

The size of the bulk buffer itself is internally determined and cannot be configured. However, the overall size of a transfer may be limited using the DB_ENV->rep_set_limit() method.

Transactional guarantees

It is important to consider replication in the context of the overall database environment's transactional guarantees. To briefly review, transactional guarantees in a non-replicated application are based on the writing of log file records to "stable storage", usually a disk drive. If the application or system then fails, the Berkeley DB logging information is reviewed during recovery, and the databases are updated so that all changes made as part of committed transactions appear, and all changes made as part of uncommitted transactions do not appear. In this case, no information will have been lost.

If a database environment does not require the log be flushed to stable storage on transaction commit (using the DB_TXN_NOSYNC flag to increase performance at the cost of sacrificing transactional durability), Berkeley DB recovery will only be able to restore the system to the state of the last commit found on stable storage. In this case, information may have been lost (for example, the changes made by some committed transactions may not appear in the databases after recovery).

Further, if there is database or log file loss or corruption (for example, if a disk drive fails), then catastrophic recovery is necessary, and Berkeley DB recovery will only be able to restore the system to the state of the last archived log file. In this case, information may also have been lost.

Replicating the database environment extends this model, by adding a new component to "stable storage": the client's replicated information. If a database environment is replicated, there is no lost information in the case of database or log file loss, because the replicated system can be configured to contain a complete set of databases and log records up to the point of failure. A database environment that loses a disk drive can have the drive replaced, and it can then rejoin the replication group.

Because of this new component of stable storage, specifying DB_TXN_NOSYNC in a replicated environment no longer sacrifices durability, as long as one or more clients have acknowledged receipt of the messages sent by the master. Since network connections are often faster than local synchronous disk writes, replication becomes a way for applications to significantly improve their performance as well as their reliability.

The return status from the application's **send** function must be set by the application to ensure the transactional guarantees the application wants to provide. Whenever the **send** function

returns failure, the local database environment's log is flushed as necessary to ensure that any information critical to database integrity is not lost. Because this flush is an expensive operation in terms of database performance, applications should avoid returning an error from the **send** function, if at all possible.

The only interesting message type for replication transactional guarantees is when the application's **send** function was called with the DB_REP_PERMANENT flag specified. There is no reason for the **send** function to ever return failure unless the DB_REP_PERMANENT flag was specified -- messages without the DB_REP_PERMANENT flag do not make visible changes to databases, and the **send** function can return success to Berkeley DB as soon as the message has been sent to the client(s) or even just copied to local application memory in preparation for being sent.

When a client receives a DB_REP_PERMANENT message, the client will flush its log to stable storage before returning (unless the client environment has been configured with the DB_TXN_NOSYNC option). If the client is unable to flush a complete transactional record to disk for any reason (for example, there is a missing log record before the flagged message), the call to the DB_ENV->rep_process_message() method on the client will return DB_REP_NOTPERM and return the LSN of this record to the application in the **ret_lsnp** parameter. The application's client or master message handling loops should take proper action to ensure the correct transactional guarantees in this case. When missing records arrive and allow subsequent processing of previously stored permanent records, the call to the DB_ENV->rep_process_message() method on the client will return DB_REP_ISPERM and return the largest LSN of the permanent records that were flushed to disk. Client applications can use these LSNs to know definitively if any particular LSN is permanently stored or not.

An application relying on a client's ability to become a master and guarantee that no data has been lost will need to write the **send** function to return an error whenever it cannot guarantee the site that will win the next election has the record. Applications not requiring this level of transactional guarantees need not have the **send** function return failure (unless the master's database environment has been configured with DB_TXN_NOSYNC), as any information critical to database integrity has already been flushed to the local log before **send** was called.

To sum up, the only reason for the **send** function to return failure is when the master database environment has been configured to not synchronously flush the log on transaction commit (that is, DB_TXN_NOSYNC was configured on the master), the DB_REP_PERMANENT flag is specified for the message, and the **send** function was unable to determine that some number of clients have received the current message (and all messages preceding the current message). How many clients need to receive the message before the **send** function can return success is an application choice (and may not depend as much on a specific number of clients reporting success as one or more geographically distributed clients).

If, however, the application does require on-disk durability on the master, the master should be configured to synchronously flush the log on commit. If clients are not configured to synchronously flush the log, that is, if a client is running with DB_TXN_NOSYNC configured, then it is up to the application to reconfigure that client appropriately when it becomes a master. That is, the application must explicitly call DB_ENV->set_flags() to disable asynchronous log flushing as part of re-configuring the client as the new master.

Of course, it is important to ensure that the replicated master and client environments are truly independent of each other. For example, it does not help matters that a client has acknowledged receipt of a message if both master and clients are on the same power supply, as the failure of the power supply will still potentially lose information.

Configuring your replication-based application to achieve the proper mix of performance and transactional guarantees can be complex. In brief, there are a few controls an application can set to configure the guarantees it makes: specification of DB_TXN_NOSYNC for the master environment, specification of DB_TXN_NOSYNC for the client environment, the priorities of different sites participating in an election, and the behavior of the application's **send** function.

Applications using Replication Manager are free to use DB_TXN_NOSYNC at the master and/or clients as they see fit. The behavior of the **send** function that Replication Manager provides on the application's behalf is determined by an "acknowledgement policy", which is configured by the DB_ENV->repmgr_set_ack_policy() method. Clients always send acknowledgements for DB_REP_PERMANENT messages (unless the acknowledgement policy in effect indicates that the master doesn't care about them). For a DB_REP_PERMANENT message, the master blocks the sending thread until either it receives the proper number of acknowledgements, or the DB_REP_ACK_TIMEOUT expires. In the case of timeout, Replication Manager returns an error code from the **send** function, causing Berkeley DB to flush the transaction log before returning to the application, as previously described. The default acknowledgement policy is DB_REPMGR_ACKS_QUORUM, which ensures that the effect of a permanent record remains durable following an election.

First, it is rarely useful to write and synchronously flush the log when a transaction commits on a replication client. It may be useful where systems share resources and multiple systems commonly fail at the same time. By default, all Berkeley DB database environments, whether master or client, synchronously flush the log on transaction commit or prepare. Generally, replication masters and clients turn log flush off for transaction commit using the DB_TXN_NOSYNC flag.

Consider two systems connected by a network interface. One acts as the master, the other as a read-only client. The client takes over as master if the master crashes and the master rejoins the replication group after such a failure. Both master and client are configured to not synchronously flush the log on transaction commit (that is, DB_TXN_NOSYNC was configured on both systems). The application's **send** function never returns failure to the Berkeley DB library, simply forwarding messages to the client (perhaps over a broadcast mechanism), and always returning success. On the client, any DB_REP_NOTPERM returns from the client's DB_ENV->rep_process_message() method are ignored, as well. This system configuration has excellent performance, but may lose data in some failure modes.

If both the master and the client crash at once, it is possible to lose committed transactions, that is, transactional durability is not being maintained. Reliability can be increased by providing separate power supplies for the systems and placing them in separate physical locations.

If the connection between the two machines fails (or just some number of messages are lost), and subsequently the master crashes, it is possible to lose committed transactions. Again, transactional durability is not being maintained. Reliability can be improved in a couple of ways:

1. Use a reliable network protocol (for example, TCP/IP instead of UDP).

2. Increase the number of clients and network paths to make it less likely that a message will be lost. In this case, it is important to also make sure a client that did receive the message wins any subsequent election. If a client that did not receive the message wins a subsequent election, data can still be lost.

Further, systems may want to guarantee message delivery to the client(s) (for example, to prevent a network connection from simply discarding messages). Some systems may want to ensure clients never return out-of-date information, that is, once a transaction commit returns success on the master, no client will return old information to a read-only query. Some of the following changes to a Base API application may be used to address these issues:

1. Write the application's **send** function to not return to Berkeley DB until one or more clients have acknowledged receipt of the message. The number of clients chosen will be dependent on the application: you will want to consider likely network partitions (ensure that a client at each physical site receives the message) and geographical diversity (ensure that a client on each coast receives the message).

2. Write the client's message processing loop to not acknowledge receipt of the message until a call to the DB_ENV->rep_process_message() method has returned success. Messages resulting in a return of DB_REP_NOTPERM from the DB_ENV->rep_process_message() method mean the message could not be flushed to the client's disk. If the client does not acknowledge receipt of such messages to the master until a subsequent call to the DB_ENV->rep_process_message() method returns DB_REP_ISPERM and the LSN returned is at least as large as this message's LSN, then the master's **send** function will not return success to the Berkeley DB library. This means the thread committing the transaction on the master will not be allowed to proceed based on the transaction having committed until the selected set of clients have received the message and consider it complete.

   Alternatively, the client's message processing loop could acknowledge the message to the master, but with an error code indicating that the application's **send** function should not return to the Berkeley DB library until a subsequent acknowledgement from the same client indicates success.

   The application send callback function invoked by Berkeley DB contains an LSN of the record being sent (if appropriate for that record). When DB_ENV->rep_process_message() method returns indicators that a permanent record has been written then it also returns the maximum LSN of the permanent record written.

There is one final pair of failure scenarios to consider. First, it is not possible to abort transactions after the application's **send** function has been called, as the master may have already written the commit log records to disk, and so abort is no longer an option. Second, a related problem is that even though the master will attempt to flush the local log if the **send** function returns failure, that flush may fail (for example, when the local disk is full). Again, the transaction cannot be aborted as one or more clients may have committed the transaction even if **send** returns failure. Rare applications may not be able to tolerate these unlikely failure modes. In that case the application may want to:

1. Configure the master to do always local synchronous commits (turning off the DB_TXN_NOSYNC configuration). This will decrease performance significantly, of course (one of the reasons to use replication is to avoid local disk writes.) In this configuration, failure to write the local log will cause the transaction to abort in all cases.

2. Do not return from the application's **send** function under any conditions, until the selected set of clients has acknowledged the message. Until the **send** function returns to the Berkeley DB library, the thread committing the transaction on the master will wait, and so no application will be able to act on the knowledge that the transaction has committed.

Master Leases

Some applications have strict requirements about the consistency of data read on a master site. Berkeley DB provides a mechanism called master leases to provide such consistency. Without master leases, it is sometimes possible for Berkeley DB to return old data to an application when newer data is available due to unfortunate scheduling as illustrated below:

1. **Application on master site**: Read data item *foo* via Berkeley DB DB->get() or DBC->get() call.

2. **Application on master site**: sleep, get descheduled, etc.

3. **System**: Master changes role, becomes a client.

4. **System**: New site is elected master.

5. **System**: New master modifies data item *foo*.

6. **Application**: Berkeley DB returns old data for *foo* to application.

By using master leases, Berkeley DB can provide guarantees about the consistency of data read on a master site. The master site can be considered a recognized authority for the data and consequently can provide authoritative reads. Clients grant master leases to a master site. By doing so, clients acknowledge the right of that site to retain the role of master for a period of time. During that period of time, clients cannot elect a new master, become master, nor grant their lease to another site.

By holding a collection of granted leases, a master site can guarantee to the application that the data returned is the current, authoritative value. As a master performs operations, it continually requests updated grants from the clients. When a read operation is required, the master guarantees that it holds a valid collection of lease grants from clients before returning data to the application. By holding leases, Berkeley DB provides several guarantees to the application:

1. Authoritative reads: A guarantee that the data being read by the application is the current value.

2. Durability from rollback: A guarantee that the data being written or read by the application is permanent across a majority of client sites and will never be rolled back.

The rollback guarantee also depends on the DB_TXN_NOSYNC flag. The guarantee is effective as long as there isn't total replication group failure while clients have granted leases but are holding the updates in their cache. The application must weigh the performance impact of synchronous transactions against the risk of total replication group failure. If clients grant a lease while holding updated data in cache, and total failure occurs, then the data is no longer present on the clients and rollback can occur if the master also crashes.

The guarantee that data will not be rolled back applies only to data successfully committed on a master. Data read on a client, or read while ignoring leases can be rolled back.

3. Freshness: A guarantee that the data being read by the application on the *master* is up-to-date and has not been modified or removed during the read.

   The read authority is only on the master. Read operations on a client always ignore leases and consequently, these operations can return stale data.

4. Master viability: A guarantee that a current master with valid leases cannot encounter a duplicate master situation.
   Leases remove the possibility of a duplicate master situation that forces the current master to downgrade to a client. However, it is still possible that old masters with expired leases can discover a later master and return DB_REP_DUPMASTER to the application.

There are several requirements of the application using leases:

1. Replication Manager applications must configure a majority (or larger) acknowledgement policy via the DB_ENV->repmgr_set_ack_policy() method. Base API applications must implement and enforce such a policy on their own.

2. Base API applications must return an error from the send callback function when the majority acknowledgement policy is not met for permanent records marked with DB_REP_PERMANENT. Note that the Replication Manager automatically fulfills this requirement.

3. Applications must set the number of sites in the group using the DB_ENV->rep_set_nsites() method before starting replication and cannot change it during operation.

4. Using leases in a replication group is all or none. Behavior is undefined when some sites configure leases and others do not. Use the DB_ENV->rep_set_config() method to turn on leases.

5. The configured lease timeout value must be the same on all sites in a replication group, set via the DB_ENV->rep_set_timeout() method.

6. The configured clock_scale_factor value must be the same on all sites in a replication group. This value defaults to no skew, but can be set via the DB_ENV->rep_set_clockskew() method.

7. Applications that care about read guarantees must perform all read operations on the master. Reading on a client does not guarantee freshness.

8. The application must use elections to choose a master site. It must never simply declare a master without having won an election (as is allowed without Master Leases).

Master leases are based on timeouts. Berkeley DB assumes that time always runs forward. Users who change the system clock on either client or master sites when leases are in use void all guarantees and can get undefined behavior. See the DB_ENV->rep_set_timeout() method for more information.

Read operations on a master that should not be subject to leases can use the DB_IGNORE_LEASE flag to the DB->get() method. Read operations on a client always imply leases are ignored.

Clients are forbidden from participating in elections while they have an outstanding lease granted to a master. Therefore, if the DB_ENV->rep_elect() method is called, then Berkeley DB will block, waiting until its lease grant expires before participating in any election. While it waits, the client attempts to contact the current master. If the client finds a current master, then it returns from the DB_ENV->rep_elect() method. When leases are configured and the lease has never yet been granted (on start-up), clients must wait a full lease timeout before participating in an election.

Clock Skew

Since master leases take into account a timeout that is used across all sites in a replication group, leases must also take into account any known skew (or drift) between the clocks on different machines in the group. The guarantees provided by master leases take clock skew into account. Consider a replication group where a client's clock is running faster than the master's clock and the group has a lease timeout of 5 seconds. If clock skew is not taken into account, eventually, the client will believe that 5 seconds have passed faster than the master and that client may then grant its lease to another site. Meanwhile, the master site does not believe 5 seconds have passed because its clock is slower, and it believes it still holds a valid lease grant. For this reason, Berkeley DB compensates for clock skew.

The master of a group using leases must account for skew in case that site has the slowest clock in the group. This computation avoids the problem of a master site believing a lease grant is valid too long. Clients in a group must account for skew in case they have the fastest clock in the group. This computation avoids the problem of a client site expiring its grant too soon and potentially granting a lease to a different site. Berkeley DB uses a conservative computation and accounts for clock skew on both sides, yielding a double compensation.

The DB_ENV->rep_set_clockskew() method takes the values for both the fastest and slowest clocks in the entire replication group as parameters. The values passed in must be the same for all sites in the group. If the user knows the maximum clock drift of their sites, then those values can be expressed as a relative percentage. Or, if the user runs an experiment then the actual values can be used.

For example, suppose the user knows that there is a maximum drift rate of 2% among sites in the group. The user should pass in 102 and 100 for the fast and slow clock values respectively. That is an unusually large value, so suppose, for example, the rate is 0.03% among sites in the group. The user should pass in 10003 and 10000 for the fast and slow clock values. Those values can be used to express the level of precision the user needs.

An example of an experiment a user can run to help determine skew would be to write a program that started simultaneously on all sites in the group. Suppose, after 1 day (86400 seconds), one site shows 86400 seconds and the other site shows it ran faster and it indicates

86460 seconds has passed. The user can use 86460 and 86400 for their parameters for the fast and slow clock values.

Since Berkeley DB is using those fast and slow clock values to compute a ratio internally, if the user cannot detect or measure any clock skew, then the same value should be passed in for both parameters, such as 1 and 1.

Network partitions

The Berkeley DB replication implementation can be affected by network partitioning problems.

For example, consider a replication group with N members. The network partitions with the master on one side and more than N/2 of the sites on the other side. The sites on the side with the master will continue forward, and the master will continue to accept write queries for the databases. Unfortunately, the sites on the other side of the partition, realizing they no longer have a master, will hold an election. The election will succeed as there are more than N/2 of the total sites participating, and there will then be two masters for the replication group. Since both masters are potentially accepting write queries, the databases could diverge in incompatible ways.

If multiple masters are ever found to exist in a replication group, a master detecting the problem will return DB_REP_DUPMASTER. If the application sees this return, it should reconfigure itself as a client (by calling DB_ENV->rep_start()), and then call for an election (by calling DB_ENV->rep_elect()). The site that wins the election may be one of the two previous masters, or it may be another site entirely. Regardless, the winning system will bring all of the other systems into conformance.

As another example, consider a replication group with a master environment and two clients A and B, where client A may upgrade to master status and client B cannot. Then, assume client A is partitioned from the other two database environments, and it becomes out-of-date with respect to the master. Then, assume the master crashes and does not come back on-line. Subsequently, the network partition is restored, and clients A and B hold an election. As client B cannot win the election, client A will win by default, and in order to get back into sync with client B, possibly committed transactions on client B will be unrolled until the two sites can once again move forward together.

In both of these examples, there is a phase where a newly elected master brings the members of a replication group into conformance with itself so that it can start sending new information to them. This can result in the loss of information as previously committed transactions are unrolled.

In architectures where network partitions are an issue, applications may want to implement a heart-beat protocol to minimize the consequences of a bad network partition. As long as a master is able to contact at least half of the sites in the replication group, it is impossible for there to be two masters. If the master can no longer contact a sufficient number of systems, it should reconfigure itself as a client, and hold an election. Replication Manager does not currently implement such a feature, so this technique is only available to Base API applications.

There is another tool applications can use to minimize the damage in the case of a network partition. By specifying an **nsites** argument to DB_ENV->rep_elect() that is larger than the

actual number of database environments in the replication group, Base API applications can keep systems from declaring themselves the master unless they can talk to a large percentage of the sites in the system. For example, if there are 20 database environments in the replication group, and an argument of 30 is specified to the DB_ENV->rep_elect() method, then a system will have to be able to talk to at least 16 of the sites to declare itself the master.

Replication Manager uses the value of **nsites** (configured by the DB_ENV->rep_set_nsites() method) for elections as well as in calculating how many acknowledgements to wait for when sending a DB_REP_PERMANENT message. So this technique may be useful here as well, unless the application uses the DB_REPMGR_ACKS_ALL or DB_REPMGR_ACKS_ALL_PEERS acknowledgement policies.

Specifying a **nsites** argument to DB_ENV->rep_elect() that is smaller than the actual number of database environments in the replication group has its uses as well. For example, consider a replication group with 2 environments. If they are partitioned from each other, neither of the sites could ever get enough votes to become the master. A reasonable alternative would be to specify a **nsites** argument of 2 to one of the systems and a **nsites** argument of 1 to the other. That way, one of the systems could win elections even when partitioned, while the other one could not. This would allow one of the systems to continue accepting write queries after the partition.

In a 2-site group, Replication Manager by default reacts to the loss of communication with the master by assuming the master has crashed: the surviving client simply declares itself to be master. Thus it avoids the problem of the survivor never being able to get enough votes to prevail. But it does leave the group vulnerable to the risk of multiple masters, if both sites are running but cannot communicate. If this risk of multiple masters is unacceptable, there is a configuration option to observe a strict majority rule that prevents the survivor from taking over. See the DB_ENV->rep_set_config() method DB_REPMGR_CONF_2SITE_STRICT flag for more information.

These scenarios stress the importance of good network infrastructure in Berkeley DB replicated environments. When replicating database environments over sufficiently lossy networking, the best solution may well be to pick a single master, and only hold elections when human intervention has determined the selected master is unable to recover at all.

Replication FAQ

1. **Does Berkeley DB provide support for forwarding write queries from clients to masters?**

   No, it does not. In general this protocol is left entirely to the application. Note, there is no reason not to use the communications channels a Base API application establishes for replication support to forward database update messages to the master, since Berkeley DB does not require those channels to be used exclusively for replication messages. Replication Manager does not currently offer this service to the application.

2. **Can I use replication to partition my environment across multiple sites?**

   No, this is not possible. All replicated databases must be equally shared by all environments in the replication group.

3. **I'm running with replication but I don't see my databases on the client.**

   This problem may be the result of the application using absolute path names for its databases, and the pathnames are not valid on the client system.

4. **How can I distinguish Berkeley DB messages from application messages?**

   There is no way to distinguish Berkeley DB messages from application-specific messages, nor does Berkeley DB offer any way to wrap application messages inside of Berkeley DB messages. Distributed applications exchanging their own messages should either enclose Berkeley DB messages in their own wrappers, or use separate network connections to send and receive Berkeley DB messages. The one exception to this rule is connection information for new sites; Berkeley DB offers a simple method for sites joining replication groups to send connection information to the other database environments in the group (see Connecting to a new site (page 198) for more information).

5. **How should I build my send function?**

   This depends on the specifics of the application. One common way is to write the **rec** and **control** arguments' sizes and data to a socket connected to each remote site. On a fast, local area net, the simplest method is likely to be to construct broadcast messages. Each Berkeley DB message would be encapsulated inside an application specific message, with header information specifying the intended recipient(s) for the message. This will likely require a global numbering scheme, however, as the Berkeley DB library has to be able to send specific log records to clients apart from the general broadcast of new log records intended for all members of a replication group.

6. **Does every one of my threads of control on the master have to set up its own connection to every client? And, does every one of my threads of control on the client have to set up its own connection to every master?**

   This is not always necessary. In the Berkeley DB replication model, any thread of control which modifies a database in the master environment must be prepared to send a message to the client environments, and any thread of control which delivers a message to a client environment must be prepared to send a message to the master. There are many ways in which these requirements can be satisfied.

   The simplest case is probably a single, multithreaded process running on the master and clients. The process running on the master would require a single write connection to each client and a single read connection from each client. A process running on each client would require a single read connection from the master and a single write connection to the master. Threads running in these processes on the master and clients would use the same network connections to pass messages back and forth.

   A common complication is when there are multiple processes running on the master and clients. A straight-forward solution is to increase the numbers of connections on the master — each process running on the master has its own write connection to each client. However, this requires only one additional connection for each possible client in the master process. The master environment still requires only a single read connection from each client (this can be done by allocating a separate thread of control which does nothing other than receive

client messages and forward them into the database). Similarly, each client still only requires a single thread of control that receives master messages and forwards them into the database, and which also takes database messages and forwards them back to the master. This model requires the networking infrastructure support many-to-one writers-to-readers, of course.

If the number of network connections is a problem in the multiprocess model, and inter-process communication on the system is inexpensive enough, an alternative is have a single process which communicates between the master and each client, and whenever a process' **send** function is called, the process passes the message to the communications process which is responsible for forwarding the message to the appropriate client. Alternatively, a broadcast mechanism will simplify the entire networking infrastructure, as processes will likely no longer have to maintain their own specific network connections.

Ex_rep: a replication example

Ex_rep, found in the `examples_c/ex_rep` subdirectory of the Berkeley DB distribution, is a simple but complete demonstration of a replicated application. The application is a mock stock ticker. The master accepts a stock symbol and a numerical value as input, and stores this information into a replicated database; either master or clients can display the contents of the database, given an empty input line.

There are two versions of the application: ex_rep_mgr uses Replication Manager, while ex_rep_base uses the replication Base API. This is intended to demonstrate that, while the basic function of the application is the same in either case, the replication support infrastructure differs markedly.

The communication infrastructure demonstrated with ex_rep_base has the same dependencies on system networking and threading support as does the Replication Manager (see the Replication introduction (page 190)). The Makefile created by the standard UNIX configuration will build the ex_rep examples on most platforms. Enter "make ex_rep_mgr" and/or "make ex_rep_base" to build them.

The synopsis for both programs is as follows:
`ex_rep_xxx` **`-h home -l host:port`** [**`-MC`**] [**`-r host:port`**] [**`-R host:port`**] [**`-a all|quorum`**] [**`-b`**] [**`-n sites`**] [**`-p priority`**] [**`-v`**]

where "ex_rep_xxx" is either "ex_rep_mgr" or "ex_rep_base". The only difference is that specifying **-M** or **-C** is optional for ex_rep_mgr, but one of these options must be specified for ex_rep_base.

The options apply to either version of the program except where noted. They are as follows:

-h

Specify a home directory for the database environment.

-l

Listen on local host "host" at port "port" for incoming connections.

-M

Configure this process as a master.

-C

Configure this process as a client.

-r

Attempt to connect to a remote member of the replication group which is listening on host "host" at port "port". Members of a replication group should be able to find all other members of a replication group so long as they are in contact with at least one other member of the replication group. Any number of these may be specified.

-R

Attempt to connect as a remote peer to a remote member of the replication group which is listening on host "host" at port "port". At most, one of these may be specified. See Client-to-client synchronization (page 203) for more information (ex_rep_mgr only.)

-a

Specify repmgr acknowledgement policy of all or quorum. See DB_ENV->repmgr_set_ack_policy() for more information (ex_rep_mgr only.)

-b

Indicates that bulk transfer should be used. See Bulk transfer (page 205) for more information.

-n

Specify the total number of sites in the replication group.

-p

Set the election priority. See Elections (page 201) for more information.

-v

Indicates that additional informational and debugging output should be enabled.

A typical ex_rep session begins with a command such as the following, to start a master:

```
ex_rep_mgr -M -p 100 -n 4 -h DIR1 -l localhost:6000
```

and several clients:

```
ex_rep_mgr -C -p 50 -n 4 -h DIR2 -l localhost:6001 -r localhost:6000
ex_rep_mgr -C -p 10 -n 4 -h DIR3 -l localhost:6002 -r localhost:6000
ex_rep_mgr -C -p  0 -n 4 -h DIR4 -l localhost:6003 -r localhost:6000
```

In this example, the client with home directory DIR4 can never become a master (its priority is 0). Both of the other clients can become masters, but the one with home directory DIR2 is preferred. Priorities are assigned by the application and should reflect the desirability of having particular clients take over as master in the case that the master fails.

Ex_rep_base: a TCP/IP based communication infrastructure

Base API applications must implement a communication infrastructure. The communication infrastructure consists of three parts: a way to map environment IDs to particular sites, the functions to get and receive messages, and the application architecture that supports the particular communication infrastructure used (for example, individual threads per communicating

site, a shared message handler for all sites, a hybrid solution). The communication infrastructure for ex_rep_base is implemented in the file ex_rep/base/rep_net.c, and each part of that infrastructure is described as follows.

Ex_rep_base maintains a table of environment ID to TCP/IP port mappings. A pointer to this table is stored in a structure pointed to by the app_private field of the DB_ENV object so it can be accessed by any function that has the database environment handle. The table is represented by a machtab_t structure which contains a reference to a linked list of member_t's, both of which are defined in ex_rep/base/rep_net.c. Each member_t contains the host and port identification, the environment ID, and a file descriptor.

This design is particular to this application and communication infrastructure, but provides an indication of the sort of functionality that is needed to maintain the application-specific state for a TCP/IP-based infrastructure. The goal of the table and its interfaces is threefold: First, it must guarantee that given an environment ID, the send function can send a message to the appropriate place. Second, when given the special environment ID DB_EID_BROADCAST, the send function can send messages to all the machines in the group. Third, upon receipt of an incoming message, the receive function can correctly identify the sender and pass the appropriate environment ID to the DB_ENV->rep_process_message() method.

Mapping a particular environment ID to a specific port is accomplished by looping through the linked list until the desired environment ID is found. Broadcast communication is implemented by looping through the linked list and sending to each member found. Since each port communicates with only a single other environment, receipt of a message on a particular port precisely identifies the sender.

This is implemented in the quote_send, quote_send_broadcast and quote_send_one functions, which can be found in ex_rep/base/rep_net.c.

The example provided is merely one way to satisfy these requirements, and there are alternative implementations as well. For instance, instead of associating separate socket connections with each remote environment, an application might instead label each message with a sender identifier; instead of looping through a table and sending a copy of a message to each member of the replication group, the application could send a single message using a broadcast protocol.

The quote_send function is passed as the callback to DB_ENV->rep_set_transport(); Berkeley DB automatically sends messages as needed for replication. The receive function is a mirror to the quote_send_one function. It is not a callback function (the application is responsible for collecting messages and calling DB_ENV->rep_process_message() on them as is convenient). In the sample application, all messages transmitted are Berkeley DB messages that get handled by DB_ENV->rep_process_message(), however, this is not always going to be the case. The application may want to pass its own messages across the same channels, distinguish between its own messages and those of Berkeley DB, and then pass only the Berkeley DB ones to DB_ENV->rep_process_message().

The final component of the communication infrastructure is the process model used to communicate with all the sites in the replication group. Each site creates a thread of control that listens on its designated socket (as specified by the **-l** command line argument) and then creates a new channel for each site that contacts it. In addition, each site explicitly connects to the sites specified in the **-r** and **-R** command line arguments. This is a fairly standard TCP/IP

process architecture and is implemented by the connect_thread, connect_all and connect_site functions in ex_rep/base/rep_msg.c and supporting functions in ex_rep/base/rep_net.c.

Ex_rep_base: putting it all together

Beyond simply initializing a replicated environment, a Base API application must set up its communication infrastructure, and then make sure that incoming messages are received and processed.

To initialize replication, ex_rep_base creates a Berkeley DB environment and calls DB_ENV->rep_set_transport() to establish a send function. (See the main function in ex_rep/base/rep_base.c, including its calls to the create_env and env_init functions in ex_rep/common/rep_common.c.)

ex_rep_base opens a listening socket for incoming connections and opens an outgoing connection to every machine that it knows about (that is, all the sites listed in the **-r** and **-R** command line arguments). Applications can structure the details of this in different ways, but ex_rep_base creates a user-level thread to listen on its socket, plus a thread to loop and handle messages on each socket, in addition to the threads needed to manage the user interface, update the database on the master, and read from the database on the client (in other words, in addition to the normal functionality of any database application).

Once the initial threads have all been started and the communications infrastructure is initialized, the application signals that it is ready for replication and joins a replication group by calling DB_ENV->rep_start(). (Again, see the main function in ex_rep/base/rep_base.c.)

Note the use of the optional second argument to DB_ENV->rep_start() in the client initialization code. The argument "local" is a piece of data, opaque to Berkeley DB, that will be broadcast to each member of a replication group; it allows new clients to join a replication group, without knowing the location of all its members; the new client will be contacted by the members it does not know about, who will receive the new client's contact information that was specified in "myaddr." See Connecting to a new site (page 198) for more information.

The final piece of a replicated application is the code that loops, receives, and processes messages from a given remote environment. ex_rep_base runs one of these loops in a parallel thread for each socket connection (see the hm_loop function in ex_rep/base/rep_msg.c). Other applications may want to queue messages somehow and process them asynchronously, or select() on a number of sockets and either look up the correct environment ID for each or encapsulate the ID in the communications protocol.

Chapter 13. Application Specific Logging and Recovery

Introduction to application specific logging and recovery

It is possible to use the Locking, Logging and Transaction subsystems of Berkeley DB to provide transaction semantics on objects other than those described by the Berkeley DB access methods. In these cases, the application will need application-specific logging and recovery functions.

For example, consider an application that provides transaction semantics on data stored in plain text files accessed using the POSIX read and write system calls. The read and write operations for which transaction protection is desired will be bracketed by calls to the standard Berkeley DB transactional interfaces, DB_ENV->txn_begin() and DB_TXN->commit(), and the transaction's locker ID will be used to acquire relevant read and write locks.

Before data is accessed, the application must make a call to the lock manager, DB_ENV->lock_get(), for a lock of the appropriate type (for example, read) on the object being locked. The object might be a page in the file, a byte, a range of bytes, or some key. It is up to the application to ensure that appropriate locks are acquired. Before a write is performed, the application should acquire a write lock on the object by making an appropriate call to the lock manager, DB_ENV->lock_get(). Then, the application should make a call to the log manager, via the automatically-generated log-writing function described as follows. This record should contain enough information to redo the operation in case of failure after commit and to undo the operation in case of abort.

When designing applications that will use the log subsystem, it is important to remember that the application is responsible for providing any necessary structure to the log record. For example, the application must understand what part of the log record is an operation code, what part identifies the file being modified, what part is redo information, and what part is undo information.

After the log message is written, the application may issue the write system call. After all requests are issued, the application may call DB_TXN->commit(). When DB_TXN->commit() returns, the caller is guaranteed that all necessary log writes have been written to disk.

At any time before issuing a DB_TXN->commit(), the application may call DB_TXN->abort(), which will result in restoration of the database to a consistent pretransaction state. (The application may specify its own recovery function for this purpose using the DB_ENV->set_app_dispatch() method. The recovery function must be able to either reapply or undo the update depending on the context, for each different type of log record. The recovery functions must not use Berkeley DB methods to access data in the environment as there is no way to coordinate these accesses with either the aborting transaction or the updates done by recovery or replication.)

If the application crashes, the recovery process uses the log to restore the database to a consistent state.

Berkeley DB includes tools to assist in the development of application-specific logging and recovery. Specifically, given a description of information to be logged in a family of log records, these tools will automatically create log-writing functions (functions that marshall their arguments into a single log record), log-reading functions (functions that read a log record and unmarshall it into a structure containing fields that map into the arguments written to the log), log-printing functions (functions that print the contents of a log record for debugging), and templates for recovery functions (functions that review log records during transaction abort or recovery). The tools and generated code are C-language and POSIX-system based, but the generated code should be usable on any system, not just POSIX systems.

A sample application that does application-specific recovery is included in the Berkeley DB distribution, in the directory `examples_c/ex_apprec`.

Defining application-specific log records

By convention, log records are described in files named `XXX.src`, where "XXX" is typically a descriptive name for a subsystem or other logical group of logging functions. These files contain interface definition language descriptions for each type of log record that is used by the subsystem.

All blank lines and lines beginning with a hash ("#") character in the XXX.src files are ignored.

The first non-comment line in the file should begin with the keyword PREFIX, followed by a string that will be prepended to every generated function name. Frequently, the PREFIX is either identical or similar to the name of the `XXX.src` file. For example, the Berkeley DB application-specific recovery example uses the file `ex_apprec.src`, which begins with the following PREFIX line:

```
PREFIX ex_apprec
```

Following the PREFIX line are the include files required by the automatically generated functions. The include files should be listed in order, prefixed by the keyword INCLUDE. For example, the Berkeley DB application-specific recovery example lists the following include files:

```
INCLUDE #include "ex_apprec.h"
```

The rest of the XXX.src file consists of log record descriptions. Each log record description begins with one of the following lines:

```
BEGIN RECORD_NAME DB_VERSION_NUMBER RECORD_NUMBER
```

```
BEGIN_COMPAT RECORD_NAME DB_VERSION_NUMBER RECORD_NUMBER
```

and ends with the line:

```
END
```

The *BEGIN* line should be used for most record types.

The *BEGIN_COMPAT* is used for log record compatibility to facilitate online upgrades of replication groups. Records created with this keyword will produce reading and printing routines,

but no logging routines. The recovery routines are retrieved from older releases, so no recovery templates will be generated for these records.

The *DB_VERSION_NUMBER* variable should be replaced with the current major and minor version of Berkeley DB, with all punctuation removed. For example, Berkeley DB version 4.2 should be 42, version 4.5 should be 45.

The *RECORD_NAME* variable should be replaced with a record name for this log record. The *RECORD_NUMBER* variable should be replaced with a record number.

The combination of PREFIX name and *RECORD_NAME*, and the *RECORD_NUMBER* must be unique for the application, that is, values for application-specific and Berkeley DB log records may not overlap. Further, because record numbers are stored in log files, which are usually portable across application and Berkeley DB releases, any change to the record numbers or log record format or should be handled as described in the Upgrading Berkeley DB installations (page 332) section on log format changes. The record number space below 10,000 is reserved for Berkeley DB itself; applications should choose record number values equal to or greater than 10,000.

Between the BEGIN and END keywords there should be one optional *DUPLICATE* line and one line for each data item logged as part of this log record.

The *DUPLICATE* line is of the form:

```
DUPLICATE RECORD_NAME DB_VERSION_NUMBER RECORD_NUMBER
```

The *DUPLICATE* specifier should be used when creating a record that requires its own record number but can use the argument structure, reading and printing routines from another record. In this case, we will create a new log record type, but use the enclosing log record type for the argument structure and the log reading and printing routines.

The format of lines for each data item logged is as follows:

```
ARG | DBT | POINTER variable_name variable_type printf_format
```

The keyword ARG indicates that the argument is a simple parameter of the type specified. For example, a file ID might be logged as:

```
ARG fileID int d
```

The keyword DBT indicates that the argument is a Berkeley DB DBT structure, containing a length and pointer to a byte string. The keyword POINTER indicates that the argument is a pointer to the data type specified (of course the data type, not the pointer, is what is logged).

The *variable_name* is the field name within the structure that will be used to refer to this item. The *variable_type* is the C-language type of the variable, and the printf format is the C-language format string, without the leading percent ("%") character, that should be used to display the contents of the field (for example, "s" for string, "d" for signed integral type, "u" for unsigned integral type, "ld" for signed long integral type, "lu" for long unsigned integral type, and so on).

For example, ex_apprec.src defines a single log record type, used to log a directory name that has been stored in a DBT:

```
BEGIN mkdir  10000
DBT dirname  DBT  s
END
```

As the name suggests, this example of an application-defined log record will be used to log the creation of a directory. There are many more examples of XXX.src files in the Berkeley DB distribution. For example, the file btree/btree.src contains the definitions for the log records supported by the Berkeley DB Btree access method.

Automatically generated functions

The XXX.src file is processed using the gen_rec.awk script included in the dist directory of the Berkeley DB distribution. This is an awk script that is executed from with the following command line:

```
awk -f gen_rec.awk \
 -v source_file=C_FILE \
 -v header_file=H_FILE \
 -v print_file=P_FILE \
 -v template_file=TMP_FILE < XXX.src
```

where *C_FILE* is the name of the file into which to place the automatically generated C code, *H_FILE* is the name of the file into which to place the automatically generated data structures and declarations, *P_FILE* is the name of the file into which to place the automatically generated C code that prints log records, and *TMP_FILE* is the name of the file into which to place a template for the recovery routines.

Because the gen_rec.awk script uses sources files located relative to the Berkeley DB dist directory, it must be run from the dist directory. For example, in building the Berkeley DB logging and recovery routines for ex_apprec, the following script is used to rebuild the automatically generated files:

```
E=../examples_c/ex_apprec


cd ../../dist
awk -f gen_rec.awk \
    -v source_file=$E/ex_apprec_auto.c \
    -v header_file=$E/ex_apprec_auto.h \
    -v print_file=$E/ex_apprec_autop.c \
    -v template_file=$E/ex_apprec_template < $E/ex_apprec.src
```

For each log record description found in the XXX.src file, the following structure declarations and #defines will be created in the file *header_file*:

```
#define DB_PREFIX_RECORD_TYPE        /* Integer ID number */

typedef struct _PREFIX_RECORD_TYPE_args {
    /*
     * These three fields are generated for every record.
     */
```

```
    u_int32_t type;      /* Record type used for dispatch. */

    /*
     * Transaction handle that identifies the transaction on whose
     * behalf the record is being logged.
     */
    DB_TXN *txnid;

    /*
     * The log sequence number returned by the previous call to log_put
     * for this transaction.
     */
    DB_LSN *prev_lsn;

    /*
     * The rest of the structure contains one field for each of
     * the entries in the record statement.
     */
};
```

Thus, the auto-generated ex_apprec_mkdir_args structure looks as follows:

```
typedef struct _ex_apprec_mkdir_args {
 u_int32_t type;
 DB_TXN *txnid;
 DB_LSN prev_lsn;
 DBT dirname;
} ex_apprec_mkdir_args;
```

The template_file will contain a template for a recovery function. The recovery function is called on each record read from the log during system recovery, transaction abort, or the application of log records on a replication client, and is expected to redo or undo the operations described by that record. The details of the recovery function will be specific to the record being logged and need to be written manually, but the template provides a good starting point. (See ex_apprec_template and ex_apprec_rec.c for an example of both the template produced and the resulting recovery function.)

The template file should be copied to a source file in the application (but not the automatically generated source_file, as that will get overwritten each time gen_rec.awk is run) and fully developed there. The recovery function takes the following parameters:

dbenv

The environment in which recovery is running.

rec

The record being recovered.

lsn

The log sequence number of the record being recovered. The prev_lsn field, automatically included in every auto-generated log record, should be returned through this argument. The prev_lsn field is used to chain

log records together to allow transaction aborts; because the recovery function is the only place that a log record gets parsed, the responsibility for returning this value lies with the recovery function writer.

op

A parameter of type db_recops, which indicates what operation is being run (DB_TXN_ABORT, DB_TXN_APPLY, DB_TXN_BACKWARD_ROLL, DB_TXN_FORWARD_ROLL or DB_TXN_PRINT).

In addition to the header_file and template_file, a source_file is created, containing a log, read, recovery, and print function for each record type.

The log function marshalls the parameters into a buffer, and calls DB_ENV->log_put() on that buffer returning 0 on success and non-zero on failure. The log function takes the following parameters:

dbenv

The environment in which recovery is running.

txnid

The transaction identifier for the transaction handle returned by DB_ENV->txn_begin().

lsnp

A pointer to storage for a log sequence number into which the log sequence number of the new log record will be returned.

syncflag

A flag indicating whether the record must be written synchronously. Valid values are 0 and DB_FLUSH.

args

The remaining parameters to the log message are the fields described in the XXX.src file, in order.

The read function takes a buffer and unmarshalls its contents into a structure of the appropriate type. It returns 0 on success and non-zero on error. After the fields of the structure have been used, the pointer returned from the read function should be freed. The read function takes the following parameters:

dbenv

The environment in which recovery is running.

recbuf

A buffer.

argp

A pointer to a structure of the appropriate type.

The print function displays the contents of the record. The print function takes the same parameters as the recovery function described previously. Although some of the parameters

are unused by the print function, taking the same parameters allows a single dispatch loop to dispatch to a variety of functions. The print function takes the following parameters:

dbenv

The environment in which recovery is running.

rec

The record being recovered.

lsn

The log sequence number of the record being recovered.

op

Unused.

Finally, the source file will contain a function (named XXX_init_print, where XXX is replaced by the prefix) which should be added to the initialization part of the standalone db_printlog utility code so that utility can be used to display application-specific log records.

Application configuration

The application should include a dispatch function that dispatches to appropriate printing and/or recovery functions based on the log record type and the operation code. The dispatch function should take the same arguments as the recovery function, and should call the appropriate recovery and/or printing functions based on the log record type and the operation code. For example, the ex_apprec dispatch function is as follows:

```
int
apprec_dispatch(dbenv, dbt, lsn, op)
 DB_ENV *dbenv;
 DBT *dbt;
 DB_LSN *lsn;
 db_recops op;
{
 u_int32_t rectype;
 /* Pull the record type out of the log record. */
 memcpy(&rectype, dbt->data, sizeof(rectype));
 switch (rectype) {
 case DB_ex_apprec_mkdir:
  return (ex_apprec_mkdir_recover(dbenv, dbt, lsn, op));
 default:
  /*
   * We've hit an unexpected, allegedly user-defined record
   * type.
   */
  dbenv->errx(dbenv, "Unexpected log record type encountered");
  return (EINVAL);
 }
}
```

Applications use this dispatch function and the automatically generated functions as follows:

1. When the application starts, call the DB_ENV->set_app_dispatch() with your dispatch function.

2. Issue a DB_ENV->txn_begin() call before any operations you want to be transaction-protected.

3. Before accessing any data, issue the appropriate lock call to lock the data (either for reading or writing).

4. Before modifying any data that is transaction-protected, issue a call to the appropriate log function.

5. Call DB_TXN->commit() to cancel all of the modifications.

The recovery functions are called in the three following cases:

1. During recovery after application or system failure, with op set to DB_TXN_FORWARD_ROLL or DB_TXN_BACKWARD_ROLL.

2. During transaction abort, with op set to DB_TXN_ABORT.

3. On a replicated client to apply updates from the master, with op set to DB_TXN_APPLY.

For each log record type you declare, you must write the appropriate function to undo and redo the modifications. The shell of these functions will be generated for you automatically, but you must fill in the details.

Your code must be able to detect whether the described modifications have been applied to the data. The function will be called with the "op" parameter set to DB_TXN_ABORT when a transaction that wrote the log record aborts, with DB_TXN_FORWARD_ROLL and DB_TXN_BACKWARD_ROLL during recovery, and with DB_TXN_APPLY on a replicated client.

The actions for DB_TXN_ABORT and DB_TXN_BACKWARD_ROLL should generally be the same, and the actions for DB_TXN_FORWARD_ROLL and DB_TXN_APPLY should generally be the same. However, if the application is using Berkeley DB replication and another thread of control may be performing read operations while log records are applied on a replication client, the recovery function should perform appropriate locking during DB_TXN_APPLY operations. In this case, the recovery function may encounter deadlocks when issuing locking calls. The application should run with the deadlock detector, and the recovery function should simply return DB_LOCK_DEADLOCK (page 230) if a deadlock is detected and a locking operation fails with that error.

The DB_TXN_PRINT operation should print the log record, typically using the auto-generated print function; it is not used in the Berkeley DB library, but may be useful for debugging, as in the db_printlog utility. Applications may safely ignore this operation code, they may handle printing from the recovery function, or they may dispatch directly to the auto-generated print function.

One common way to determine whether operations need to be undone or redone is the use of log sequence numbers (LSNs). For example, each access method database page contains the LSN of the most recent log record that describes a modification to the page. When the access method changes a page, it writes a log record describing the change and including the LSN that was on the page before the change. This LSN is referred to as the previous LSN. The recovery

functions read the page described by a log record, and compare the LSN on the page to the LSN they were passed.

If the page LSN is less than the passed LSN and the operation is an undo, no action is necessary (because the modifications have not been written to the page). If the page LSN is the same as the previous LSN and the operation is a redo, the actions described are reapplied to the page. If the page LSN is equal to the passed LSN and the operation is an undo, the actions are removed from the page; if the page LSN is greater than the passed LSN and the operation is a redo, no further action is necessary. If the action is a redo and the LSN on the page is less than the previous LSN in the log record, it is an error because it could happen only if some previous log record was not processed.

Examples of other recovery functions can be found in the Berkeley DB library recovery functions (found in files named XXX_rec.c) and in the application-specific recovery example (specifically, ex_apprec_rec.c).

Finally, applications need to ensure that any data modifications they have made, that were part of a committed transaction, must be written to stable storage before calling the DB_ENV->txn_checkpoint() method. This is to allow the periodic removal of database environment log files.

Chapter 14. Programmer Notes

Signal handling

When applications using Berkeley DB receive signals, it is important that they exit gracefully, discarding any Berkeley DB locks that they may hold. This is normally done by setting a flag when a signal arrives and then checking for that flag periodically within the application. Because Berkeley DB is not re-entrant, the signal handler should not attempt to release locks and/or close the database handles itself. Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

If an application exits holding a lock, the situation is no different than if the application crashed, and all applications participating in the database environment must be shut down, and then recovery must be performed. If this is not done, databases may be left in an inconsistent state, or locks the application held may cause unresolvable deadlocks inside the environment, causing applications to hang.

Berkeley DB restarts all system calls interrupted by signals, that is, any underlying system calls that return failure with errno set to EINTR will be restarted rather than failing.

Error returns to applications

Except for the historic dbm, ndbm and hsearch interfaces, Berkeley DB does not use the global variable `errno` to return error values. The return values for all Berkeley DB functions are grouped into the following three categories:

0

A return value of 0 indicates that the operation was successful.

> 0

A return value that is greater than 0 indicates that there was a system error. The **errno** value returned by the system is returned by the function; for example, when a Berkeley DB function is unable to allocate memory, the return value from the function will be ENOMEM.

< 0

A return value that is less than 0 indicates a condition that was not a system failure, but was not an unqualified success, either. For example, a routine to retrieve a key/data pair from the database may return DB_NOTFOUND when the key/data pair does not appear in the database; as opposed to the value of 0, which would be returned if the key/data pair were found in the database.
All values returned by Berkeley DB functions are less than 0 in order to avoid conflict with possible values of **errno**. Specifically, Berkeley DB reserves all values from -30,800 to -30,999 to itself as possible error values. There are a few Berkeley DB interfaces where it is possible for an application function to be called by a Berkeley DB function and subsequently fail with an application-specific return. Such failure returns will be passed back to the function that originally called a Berkeley DB interface. To avoid ambiguity about the cause of the error, error values separate from the Berkeley DB error name space should be used.

Although possible error returns are specified by each individual function's manual page, there are a few error returns that deserve general mention:

DB_NOTFOUND and DB_KEYEMPTY

There are two special return values that are similar in meaning and that are returned in similar situations, and therefore might be confused: DB_NOTFOUND and DB_KEYEMPTY.

The DB_NOTFOUND error return indicates that the requested key/data pair did not exist in the database or that start-of- or end-of-file has been reached by a cursor.

The DB_KEYEMPTY error return indicates that the requested key/data pair logically exists but was never explicitly created by the application (the Recno and Queue access methods will automatically create key/data pairs under some circumstances; see DB->open() for more information), or that the requested key/data pair was deleted and never re-created. In addition, the Queue access method will return DB_KEYEMPTY for records that were created as part of a transaction that was later aborted and never re-created.

DB_KEYEXIST

The DB_KEYEXIST error return indicates the DB_NOOVERWRITE option was specified when inserting a key/data pair into the database and the key already exists in the database, or the DB_NODUPDATA option was specified and the key/data pair already exists in the data.

DB_LOCK_DEADLOCK

When multiple threads of control are modifying the database, there is normally the potential for deadlock. In Berkeley DB, deadlock is signified by an error return from the Berkeley DB function of the value DB_LOCK_DEADLOCK. Whenever a Berkeley DB function returns DB_LOCK_DEADLOCK, the enclosing transaction should be aborted.

Any Berkeley DB function that attempts to acquire locks can potentially return DB_LOCK_DEADLOCK. Practically speaking, the safest way to deal with applications that can deadlock is to anticipate a DB_LOCK_DEADLOCK return from any DB or DBC handle method call, or any DB_ENV handle method call that references a database, including the database's backing physical file.

DB_LOCK_NOTGRANTED

If a lock is requested from the DB_ENV->lock_get() or DB_ENV->lock_vec() methods with the DB_LOCK_NOWAIT flag specified, the method will return DB_LOCK_NOTGRANTED if the lock is not immediately available.

If the DB_TIME_NOTGRANTED flag is specified to the DB_ENV->set_flags() method, database calls timing out based on lock or transaction timeout values will return DB_LOCK_NOTGRANTED instead of DB_LOCK_DEADLOCK.

DB_RUNRECOVERY

There exists a class of errors that Berkeley DB considers fatal to an entire Berkeley DB environment. An example of this type of error is a corrupted database page. The only way to

recover from these failures is to have all threads of control exit the Berkeley DB environment, run recovery of the environment, and re-enter Berkeley DB. (It is not strictly necessary that the processes exit, although that is the only way to recover system resources, such as file descriptors and memory, allocated by Berkeley DB.)

When this type of error is encountered, the error value DB_RUNRECOVERY is returned. This error can be returned by any Berkeley DB interface. Once DB_RUNRECOVERY is returned by any interface, it will be returned from all subsequent Berkeley DB calls made by any threads of control participating in the environment.

Applications can handle such fatal errors in one of two ways: first, by checking for DB_RUNRECOVERY as part of their normal Berkeley DB error return checking, similarly to DB_LOCK_DEADLOCK or any other error. Alternatively, applications can specify a fatal-error callback function using the DB_ENV->set_event_notify() method. Applications with no cleanup processing of their own should simply exit from the callback function.

DB_SECONDARY_BAD

The DB_SECONDARY_BAD error is returned if a secondary index has been corrupted. This may be the result of an application operating on related databases without first associating them.

Environment variables

The Berkeley DB library uses the following environment variables:

DB_HOME

If the environment variable DB_HOME is set, it is used as part of File naming (page 128). Note: For the DB_HOME variable to take effect, either the DB_USE_ENVIRON or DB_USE_ENVIRON_ROOT flags must be specified to DB_ENV->open().

TMPDIR, TEMP, TMP, TempFolder

The TMPDIR, TEMP, TMP, and TempFolder environment variables are all checked as locations in which to create temporary files. See DB_ENV->set_tmp_dir() for more information.

Multithreaded applications

Berkeley DB fully supports multithreaded applications. The Berkeley DB library is not itself multithreaded, and was deliberately architected to not use threads internally because of the portability problems that would introduce. Database environment and database object handles returned from Berkeley DB library functions are free-threaded. No other object handles returned from the Berkeley DB library are free-threaded. The following rules should be observed when using threads to access the Berkeley DB library:

1. The DB_THREAD flag must be specified to the DB_ENV->open() and DB->open() methods if the Berkeley DB handles returned by those interfaces will be used in the context of more than one thread. Setting the DB_THREAD flag inconsistently may result in database corruption.

   Threading is assumed in the Java API, so no special flags are required; and Berkeley DB functions will always behave as if the DB_THREAD flag was specified.

Only a single thread may call the DB_ENV->close() or DB->close() methods for a returned environment or database handle.

No other Berkeley DB handles are free-threaded.

2. When using the non-cursor Berkeley DB calls to retrieve key/data items (for example, DB->get()), the memory to which the pointer stored into the Dbt refers is valid only until the next call using the DB handle returned by DB->open(). This includes **any** use of the returned DB handle, including by another thread within the process.

   For this reason, if the DB_THREAD handle was specified to the DB->open() method, either DB_DBT_MALLOC, DB_DBT_REALLOC or DB_DBT_USERMEM must be specified in the DBT when performing any non-cursor key or data retrieval.

3. Cursors may not span transactions. Each cursor must be allocated and deallocated within the same transaction.

   Transactions and cursors may span threads, but only serially, that is, the application must serialize access to the TXN and DBC handles. In the case of nested transactions, since all child transactions are part of the same parent transaction, they must observe the same constraints. That is, children may execute in different threads only if each child executes serially.

4. User-level synchronization mutexes must have been implemented for the compiler/architecture combination. Attempting to specify the DB_THREAD flag will fail if fast mutexes are not available.

   If blocking mutexes are available (for example POSIX pthreads), they will be used. Otherwise, the Berkeley DB library will make a system call to pause for some amount of time when it is necessary to wait on a lock. This may not be optimal, especially in a thread-only environment, in which it is usually more efficient to explicitly yield the processor to another thread.

   It is possible to specify a yield function on an per-application basis. See db_env_set_func_yield for more information.

   It is possible to specify the number of attempts that will be made to acquire the mutex before waiting. See DB_ENV->mutex_set_tas_spins() for more information.

When creating multiple databases in a single physical file, multithreaded programs may have additional requirements. For more information, see Opening multiple databases in a single file (page 40)

Berkeley DB handles

The Berkeley DB library has a number of object handles. The following table lists those handles, their scope, and whether they are free-threaded (that is, whether multiple threads within a process can share them).

DB_ENV

The DB_ENV handle, created by the db_env_create() method, refers to a Berkeley DB database environment — a collection of Berkeley DB subsystems, log files and databases. DB_ENV handles are free-threaded if the DB_THREAD flag is specified to the DB_ENV->open() method when the environment is opened. The handle should not be closed while any other handle remains open that is using it as a reference (for example, DB, TXN). Once either the DB_ENV->close() or DB_ENV->remove() methods are called, the handle may not be accessed again, regardless of the method's return.

TXN

The TXN handle, created by the DB_ENV->txn_begin() method, refers to a single transaction. The handle is not free-threaded. Transactions may span threads, but only serially, that is, the application must serialize access to the TXN handles. In the case of nested transactions, since all child transactions are part of the same parent transaction, they must observe the same constraints. That is, children may execute in different threads only if each child executes serially.

Once the DB_TXN->abort() or DB_TXN->commit() methods are called, the handle may not be accessed again, regardless of the method's return. In addition, parent transactions may not issue any Berkeley DB operations while they have active child transactions (child transactions that have not yet been committed or aborted) except for DB_ENV->txn_begin(), DB_TXN->abort() and DB_TXN->commit().

DB_LOGC

The DB_LOGC handle refers to a cursor into the log files. The handle is not free-threaded. Once the DB_LOGC->close() method is called, the handle may not be accessed again, regardless of the method's return.

DB_MPOOLFILE

The DB_MPOOLFILE handle refers to an open file in the shared memory buffer pool of the database environment. The handle is not free-threaded. Once the DB_MPOOLFILE->close() method is called, the handle may not be accessed again, regardless of the method's return.

DB

The DB handle, created by the db_create() method, refers to a single Berkeley DB database, which may or may not be part of a database environment. DB handles are free-threaded if the DB_THREAD flag is specified to the DB_ENV->open() method when the database is opened or if the database environment in which the database is opened is free-threaded. The handle should not be closed while any other handle that refers to the database is in use; for example, database handles should be left open while cursor handles into the database remain open, or transactions that include operations on the database have not yet been committed or aborted. Once the DB->close(), DB->remove() or DB->rename() methods are called, the handle may not be accessed again, regardless of the method's return.

DBC

The DBC handle refers to a cursor into a Berkeley DB database. The handle is not free-threaded. Cursors may span threads, but only serially, that is, the application must serialize access to the DBC handles. If the cursor is to be used to perform

operations on behalf of a transaction, the cursor must be opened and closed within the context of that single transaction. Once DBC->close() has been called, the handle may not be accessed again, regardless of the method's return.

Name spaces

C Language Name Space

The Berkeley DB library is careful to avoid C language programmer name spaces, but there are a few potential areas for concern, mostly in the Berkeley DB include file db.h. The db.h include file defines a number of types and strings. Where possible, all of these types and strings are prefixed with "DB_" or "db_". There are a few notable exceptions.

The Berkeley DB library uses a macro named "__P" to configure for systems that do not provide ANSI C function prototypes. This could potentially collide with other systems using a "__P" macro for similar or different purposes.

The Berkeley DB library needs information about specifically sized types for each architecture. If they are not provided by the system, they are typedef'd in the db.h include file. The types that may be typedef'd by db.h include the following: u_int8_t, int16_t, u_int16_t, int32_t, u_int32_t, u_char, u_short, u_int, and u_long.

The Berkeley DB library declares a few external routines. All these routines are prefixed with the strings "db_". All internal Berkeley DB routines are prefixed with the strings "__XXX_", where "XXX" is the subsystem prefix (for example, "__db_XXX_" and "__txn_XXX_").

Filesystem Name Space

Berkeley DB environments create or use some number of files in environment home directories. These files are named DB_CONFIG, "log.NNNNN" (for example, log.0000000003, where the number of digits following the dot is unspecified), or with the string prefix "__db" (for example, __db.001). Applications should never create files or databases in database environment home directories with names beginning with the characters "log" or "__db".

In some cases, applications may choose to remove Berkeley DB files as part of their cleanup procedures, using system utilities instead of Berkeley DB interfaces (for example, using the UNIX rm utility instead of the DB_ENV->remove() method). This is not a problem, as long as applications limit themselves to removing only files named "__db.###", where "###" are the digits 0 through 9. Applications should never remove any files named with the prefix "__db" or "log", other than "__db.###" files.

Memory-only or Flash configurations

Berkeley DB supports a variety of memory-based configurations for systems where filesystem space is either limited in availability or entirely replaced by some combination of memory and Flash. In addition, Berkeley DB can be configured to minimize writes to the filesystem when the filesystem is backed by Flash memory.

There are four parts of the Berkeley DB database environment normally written to the filesystem: the database environment region files, the database files, the database environment log files and the replication internal files. Each of these items can be configured to live in memory rather than in the filesystem:

The database environment region files:

Each of the Berkeley DB subsystems in a database environment is described by one or more regions, or chunks of memory. The regions contain all of the per-process and per-thread shared information (including mutexes), that comprise a Berkeley DB environment. By default, these regions are backed by the filesystem. In situations where filesystem backed regions aren't optimal, applications can create memory-only database environments in two different types of memory: either in the application's heap memory or in system shared memory.

To create the database environment in heap memory, specify the DB_PRIVATE flag to the DB_ENV->open() method. Note that database environments created in heap memory are only accessible to the threads of a single process, however.

To create the database environment in system shared memory, specify the DB_SYSTEM_MEM flag to the DB_ENV->open() method. Database environments created in system memory are accessible to multiple processes, but note that database environments created in system shared memory do create a small (roughly 8 byte) file in the filesystem, read by the processes to identify which system shared memory segments to use.

For more information, see Shared memory regions (page 131).

The database files:

By default, databases are periodically flushed from the Berkeley DB memory cache to backing physical files in the filesystem. To keep databases from being written to backing physical files, pass the DB_MPOOL_NOFILE flag to the DB_MPOOLFILE->set_flags() method. This flag implies the application's databases must fit entirely in the Berkeley DB cache, of course. To avoid a database file growing to consume the entire cache, applications can limit the size of individual databases in the cache by calling the DB_MPOOLFILE->set_maxsize() method.

The database environment log files:

If a database environment is not intended to be transactionally recoverable after application or system failure (that is, if it will not exhibit the transactional attribute of "durability"), applications should not configure the database environment for logging or transactions, in which case no log files will be created. If the database environment is intended to be durable, log files must either be written to stable storage and recovered after application or system failure, or they must be replicated to other systems.

In applications running on systems without any form of stable storage, durability must be accomplished through replication. In this case, database environments should be configured to maintain database logs in memory, rather than in the filesystem, by specifying the DB_LOG_IN_MEMORY flag to the DB_ENV->log_set_config() method.

The replication internal files:

By default, Berkeley DB replication stores a small amount of internal data in the filesystem. To store this replication internal data in memory only and not in the filesystem, specify the DB_REP_CONF_INMEM flag to the DB_ENV->rep_set_config() method before opening the database environment.

In systems where the filesystem is backed by Flash memory, the number of times the Flash memory is written may be a concern. Each of the four parts of the Berkeley DB database environment normally written to the filesystem can be configured to minimize the number of times the filesystem is written:

The database environment region files:

On a Flash-based filesystem, the database environment should be placed in heap or system memory, as described previously.

The database files:

The Berkeley DB library maintains a cache of database pages. The database pages are only written to backing physical files when the application checkpoints the database environment with the DB_ENV->txn_checkpoint() method, when database handles are closed with the DB->close() method, or when the application explicitly flushes the cache with the DB->sync() or DB_ENV->memp_sync() methods.

To avoid unnecessary writes of Flash memory due to checkpoints, applications should decrease the frequency of their checkpoints. This is especially important in applications which repeatedly modify a specific database page, as repeatedly writing a database page to the backing physical file will repeatedly update the same blocks of the filesystem.

To avoid unnecessary writes of the filesystem due to closing a database handle, applications should specify the DB_NOSYNC flag to the DB->close() method.

To avoid unnecessary writes of the filesystem due to flushing the cache, applications should not explicitly flush the cache under normal conditions – flushing the cache is rarely if ever needed in a normally-running application.

The database environment log files:

The Berkeley DB log files do not repeatedly overwrite the same blocks of the filesystem as the Berkeley DB log files are not implemented as a circular buffer and log files are not re-used. For this reason, the Berkeley DB log files should not cause any difficulties for Flash memory configurations.

However, as Berkeley DB does not write log records in filesystem block sized pieces, it is probable that sequential transaction commits (each of which flush the log file to the backing filesystem), will write a block of Flash memory twice, as the last log record from the first commit will write the same block of Flash memory as the first log record from the second commit. Applications not requiring absolute durability should specify the DB_TXN_WRITE_NOSYNC or DB_TXN_NOSYNC flags to the DB_ENV->set_flags() method to avoid this overwrite of a block of Flash memory.

The replication internal files:

On a Flash-based filesystem, the replication internal data should be stored in memory only, as described previously.

Disk drive caches

Many disk drives contain onboard caches. Some of these drives include battery-backup or other functionality that guarantees that all cached data will be completely written if the power fails. These drives can offer substantial performance improvements over drives without caching support. However, some caching drives rely on capacitors or other mechanisms that guarantee only that the write of the current sector will complete. These drives can endanger your database and potentially cause corruption of your data.

To avoid losing your data, make sure the caching on your disk drives is properly configured so the drive will never report that data has been written unless the data is guaranteed to be written in the face of a power failure. Many times, this means that write-caching on the disk drive must be disabled.

Copying or moving databases

There are two issues with copying or moving databases: database page log sequence numbers (LSNs), and database file identification strings.

Because database pages contain references to the database environment log records (LSNs), databases cannot be copied or moved from one transactional database environment to another without first clearing the LSNs. Note that this is not a concern for non-transactional database environments and applications, and can be ignored if the database is not being used transactionally. Specifically, databases created and written non-transactionally (for example, as part of a bulk load procedure), can be copied or moved into a transactional database environment without resetting the LSNs. The database's LSNs may be reset in one of three ways: the application can call the DB_ENV->lsn_reset() method to reset the LSNs in place, or a system administrator can reset the LSNs in place using the **-r** option to the db_load utility, or by dumping and reloading the database (using the db_dump utility and the db_load utility).

Because system file identification information (for example, filenames, device and inode numbers, volume and file IDs, and so on) are not necessarily unique or maintained across system reboots, each Berkeley DB database file contains a unique 20-byte file identification bytestring. When multiple processes or threads open the same database file in Berkeley DB, it is this bytestring that is used to ensure the same underlying pages are updated in the database environment cache, no matter which Berkeley DB handle is used for the operation.

The database file identification string is not a concern when moving databases, and databases may be moved or renamed without resetting the identification string. However, when copying a database, you must ensure there are never two databases with the same file identification bytestring in the same cache at the same time. Copying databases is further complicated because Berkeley DB caches do not discard cached database pages when database handles are closed. Cached pages are only discarded when the database is removed by calling the DB_ENV->remove() or DB->remove() methods.

Before physically copying a database file, first ensure that all modified pages have been written from the cache to the backing database file. This is done using the DB->sync() or DB->close() methods.

Before using a copy of a database file in a database environment, you must ensure that all pages from any other database with the same bytestring have been removed from the memory pool cache. If the environment in which you will open the copy of the database has pages from files with identical bytestrings to the copied database, there are a few possible solutions:

1. Remove the environment, either using system utilities or by calling the DB_ENV->remove() method. Obviously, this will not allow you to access both the original database and the copy of the database at the same time.

2. Create a new file that will have a new bytestring. The simplest way to create a new file that will have a new bytestring is to call the db_dump utility to dump out the contents of the database and then use the db_load utility to load the dumped output into a new file. This allows you to access both the original and copy of the database at the same time.

3. If your database is too large to be dumped and reloaded, you can copy the database by other means, and then reset the bytestring in the copied database to a new bytestring. There are two ways to reset the bytestring in the copy: the application can call the DB_ENV->fileid_reset() method, or a system administrator can use the **-r** option to the db_load utility. This allows you to access both the original and copy of the database at the same time.

Compatibility with historic UNIX interfaces

The Berkeley DB version 2 library provides backward-compatible interfaces for the historic UNIX dbm, ndbm and hsearch interfaces. It also provides a backward-compatible interface for the historic Berkeley DB 1.85 release.

Berkeley DB version 2 does not provide database compatibility for any of the previous interfaces, and existing databases must be converted manually. To convert existing databases from the Berkeley DB 1.85 format to the Berkeley DB version 2 format, review the db_dump185 utility and the db_load utility information. No utilities are provided to convert UNIX dbm, ndbm or hsearch databases.

Run-time configuration

It is possible for applications to configure Berkeley DB at run-time to redirect Berkeley DB library and system calls to alternate interfaces. For example, an application might want Berkeley DB to call debugging memory allocation routines rather than the standard C library interfaces. The following interfaces support this functionality:

- db_env_set_func_close
- db_env_set_func_dirfree
- db_env_set_func_dirlist

- db_env_set_func_exists
- db_env_set_func_file_map
- db_env_set_func_free
- db_env_set_func_fsync
- db_env_set_func_ftruncate
- db_env_set_func_ioinfo
- db_env_set_func_malloc
- db_env_set_func_open
- db_env_set_func_pread
- db_env_set_func_pwrite
- db_env_set_func_read
- db_env_set_func_realloc
- db_env_set_func_region_map
- db_env_set_func_rename
- db_env_set_func_seek
- db_env_set_func_unlink
- db_env_set_func_write
- db_env_set_func_yield

These interfaces are available only on POSIX platforms and from the Berkeley DB C language API.

A not-uncommon problem for applications is the new API in Solaris 2.6 for manipulating large files. Because this API was not part of Solaris 2.5, it is difficult to create a single binary that takes advantage of the large file functionality in Solaris 2.6, but still runs on Solaris 2.5. Example code [solaris.txt] that supports this is included in the Berkeley DB distribution, however, the example code was written using previous versions of the Berkeley DB APIs, and is only useful as an example.

Programmer notes FAQ

1. **What priorities should threads/tasks executing Berkeley DB functions be given?**

   Tasks executing Berkeley DB functions should have the same, or roughly equivalent, system priorities. For example, it can be dangerous to give tasks of control performing checkpoints

a lower priority than tasks of control doing database lookups, and starvation can sometimes result.

2. **Why isn't the C++ API exception safe?**

   The Berkeley DB C++ API is a thin wrapper around the C API that maps most return values to exceptions, and gives the C++ handles the same lifecycles as their C counterparts. One consequence is that if an exception occurs while a cursor or transaction handle is open, the application must explicitly close the cursor or abort the transaction.

   Applications can be simplified and bugs avoided by creating wrapper classes around DBC and TXN that call the appropriate cleanup method in the wrapper's destructor. By creating an instance of the wrappers on the stack, C++ scoping rules will ensure that the destructor is called before exception handling unrolls the block that contains the wrapper object.

Chapter 15. The Locking Subsystem

Introduction to the locking subsystem

The locking subsystem provides interprocess and intraprocess concurrency control mechanisms. Although the lock system is used extensively by the Berkeley DB access methods and transaction system, it may also be used as a standalone subsystem to provide concurrency control to any set of designated resources.

The Lock subsystem is created, initialized, and opened by calls to DB_ENV->open() with the DB_INIT_LOCK or DB_INIT_CDB flags specified.

The DB_ENV->lock_vec() method is used to acquire and release locks. The DB_ENV->lock_vec() method performs any number of lock operations atomically. It also provides the capability to release all locks held by a particular locker and release all the locks on a particular object. (Performing multiple lock operations atomically is useful in performing Btree traversals -- you want to acquire a lock on a child page and once acquired, immediately release the lock on its parent. This is traditionally referred to as *lock-coupling*). Two additional methods, DB_ENV->lock_get() and DB_ENV->lock_put(), are provided. These methods are simpler front-ends to the DB_ENV->lock_vec() functionality, where DB_ENV->lock_get() acquires a lock, and DB_ENV->lock_put() releases a lock that was acquired using DB_ENV->lock_get() or DB_ENV->lock_vec(). All locks explicitly requested by an application should be released via calls to DB_ENV->lock_put() or DB_ENV->lock_vec(). Using DB_ENV->lock_vec() instead of separate calls to DB_ENV->lock_put() and DB_ENV->lock_put() also reduces the synchronization overhead between multiple threads or processes. The three methods are fully compatible, and may be used interchangeably.

Applications must specify lockers and lock objects appropriately. When used with the Berkeley DB access methods, lockers and objects are handled completely internally, but an application using the lock manager directly must either use the same conventions as the access methods or define its own convention to which it adheres. If an application is using the access methods with locking at the same time that it is calling the lock manager directly, the application must follow a convention that is compatible with the access methods' use of the locking subsystem. See Berkeley DB Transactional Data Store locking conventions (page 253) for more information.

The DB_ENV->lock_id() function returns a unique ID that may safely be used as the locker parameter to the DB_ENV->lock_vec() method. The access methods use DB_ENV->lock_id() to generate unique lockers for the cursors associated with a database.

The DB_ENV->lock_detect() function provides the programmatic interface to the Berkeley DB deadlock detector. Whenever two threads of control issue lock requests concurrently, the possibility for deadlock arises. A deadlock occurs when two or more threads of control are blocked, waiting for actions that another one of the blocked threads must take. For example, assume that threads A and B have each obtained read locks on object X. Now suppose that both threads want to obtain write locks on object X. Neither thread can be granted its write lock (because of the other thread's read lock). Both threads block and will never unblock because the event for which they are waiting can never happen.

The deadlock detector examines all the locks held in the environment, and identifies situations where no thread can make forward progress. It then selects one of the participants in the deadlock (according to the argument that was specified to DB_ENV->set_lk_detect()), and forces it to return the value DB_LOCK_DEADLOCK (page 230), which indicates that a deadlock occurred. The thread receiving such an error must release all of its locks and undo any incomplete modifications to the locked resource. Locks are typically released, and modifications undone, by closing any cursors involved in the operation and aborting any transaction enclosing the operation. The operation may optionally be retried.

The DB_ENV->lock_stat() function returns information about the status of the lock subsystem. It is the programmatic interface used by the db_stat utility.

The locking subsystem is closed by the call to DB_ENV->close().

Finally, the entire locking subsystem may be discarded using the DB_ENV->remove() method.

Locking Subsystem and Related Methods	Description
DB_ENV->lock_detect()	Perform deadlock detection
DB_ENV->lock_get()	Acquire a lock
DB_ENV->lock_id()	Acquire a locker ID
DB_ENV->lock_id_free()	Release a locker ID
DB_ENV->lock_put()	Release a lock
DB_ENV->lock_stat()	Return lock subsystem statistics
DB_ENV->lock_vec()	Acquire/release locks
DB_ENV->cdsgroup_begin()	Get a locker ID in Berkeley DB Concurrent Data Store
Locking Subsystem Configuration	
DB_ENV->set_lk_conflicts()	Set lock conflicts matrix
DB_ENV->set_lk_detect()	Set automatic deadlock detection
DB_ENV->set_lk_max_lockers()	Set maximum number of lockers
DB_ENV->set_lk_max_locks()	Set maximum number of locks
DB_ENV->set_lk_max_objects()	Set maximum number of lock objects
DB_ENV->set_lk_partitions()	Set number of lock partitions
DB_ENV->set_timeout()	Set lock and transaction timeout

Configuring locking

The DB_ENV->set_lk_detect() method specifies that the deadlock detector should be run whenever a lock is about to block. This option provides for rapid detection of deadlocks at the expense of potentially frequent invocations of the deadlock detector. On a fast processor with a highly contentious application where response time is critical, this is a good choice. An option argument to the DB_ENV->set_lk_detect() method indicates which lock requests should be rejected.

The application can limit how long it blocks on a contested resource. The DB_ENV->set_timeout() method specifies the length of the timeout. This value is checked whenever deadlock detection is performed, so the accuracy of the timeout depends upon the frequency of deadlock detection.

In general, when applications are not specifying lock and transaction timeout values, the DB_LOCK_DEFAULT option is probably the correct first choice, and other options should only be selected based on evidence that they improve transaction throughput. If an application has long-running transactions, DB_LOCK_YOUNGEST will guarantee that transactions eventually complete, but it may do so at the expense of a large number of lock request rejections (and therefore, transaction aborts).

The alternative to using the DB_ENV->set_lk_detect() method is to explicitly perform deadlock detection using the Berkeley DB DB_ENV->lock_detect() method.

The DB_ENV->set_lk_conflicts() method allows you to specify your own locking conflicts matrix. This is an advanced configuration option, and is almost never necessary.

Configuring locking: sizing the system

The lock system is sized using the following four methods:

1. DB_ENV->set_lk_max_locks()

2. DB_ENV->set_lk_max_lockers()

3. DB_ENV->set_lk_max_objects()

4. DB_ENV->set_lk_partitions()

The DB_ENV->set_lk_max_locks(), DB_ENV->set_lk_max_lockers(), and and DB_ENV->set_lk_max_objects() methods specify the maximum number of locks, lockers, and locked objects supported by the lock subsystem, respectively. The maximum number of locks is the number of locks that can be simultaneously requested in the system. The maximum number of lockers is the number of lockers that can simultaneously request locks in the system. The maximum number of lock objects is the number of objects that can simultaneously be locked in the system. Selecting appropriate values requires an understanding of your application and its databases. If the values are too small, requests for locks in an application will fail. If the values are too large, the locking subsystem will consume more resources than is necessary. It is better to err in the direction of allocating too many locks, lockers, and objects because increasing the number of locks does not require large amounts of additional resources. The default values are 1000 of each type of object.

The DB_ENV->set_lk_partitions() method specifies the number of lock table partitions. Each partition may be accessed independently by a thread and more partitions can lead to higher levels of concurrency. The default is to set the number of partitions to be 10 times the number of cpus that the operating system reports at the time the environment is created. Having more than one partition when there is only one cpu is not beneficial and the locking system is more efficient when there is a single partition. Operating systems (Linux, Solaris) may report thread contexts as cpus and it may be necessary to override the default to force a single partition on a single hyperthreaded cpu system. Objects and locks are divided among the partitions so it

best to allocate several locks and objects per partition. The system will force there to be at least one per partition. If a partition runs out of locks or objects it will steal what is needed from the other partitions. This operation could impact performance if it occurs too often.

When configuring a Berkeley DB Concurrent Data Store application, the number of lock objects needed is two per open database (one for the database lock, and one for the cursor lock when the DB_CDB_ALLDB option is not specified). The number of locks needed is one per open database handle plus one per simultaneous cursor or non-cursor operation.

Configuring a Berkeley DB Transactional Data Store application is more complicated. The recommended algorithm for selecting the maximum number of locks, lockers, and lock objects is to run the application under stressful conditions and then review the lock system's statistics to determine the maximum number of locks, lockers, and lock objects that were used. Then, double these values for safety. However, in some large applications, finer granularity of control is necessary in order to minimize the size of the Lock subsystem.

The maximum number of lockers can be estimated as follows:

- If the database environment is using transactions, the maximum number of lockers can be estimated by adding the number of simultaneously active non-transactional cursors open database handles to the number of simultaneously active transactions and child transactions (where a child transaction is active until it commits or aborts, not until its parent commits or aborts).

- If the database environment is not using transactions, the maximum number of lockers can be estimated by adding the number of simultaneously active non-transactional cursors and open database handles to the number of simultaneous non-cursor operations.

The maximum number of lock objects needed for a single database operation can be estimated as follows:

- For Btree and Recno access methods, you will need one lock object per level of the database tree, at a minimum. (Unless keys are quite large with respect to the page size, neither Recno nor Btree database trees should ever be deeper than five levels.) Then, you will need one lock object for each leaf page of the database tree that will be simultaneously accessed.

- For the Queue access method, you will need one lock object per record that is simultaneously accessed. To this, add one lock object per page that will be simultaneously accessed. (Because the Queue access method uses fixed-length records and the database page size is known, it is possible to calculate the number of pages -- and, therefore, the lock objects -- required.) Deleted records skipped by a DB_NEXT or DB_PREV operation do not require a separate lock object. Further, if your application is using transactions, no database operation will ever use more than three lock objects at any time.

- For the Hash access method, you only need a single lock object.

For all access methods, you should then add an additional lock object per database for the database's metadata page.

Note that transactions accumulate locks over the transaction lifetime, and the lock objects required by a single transaction is the total lock objects required by all of the database

operations in the transaction. However, a database page (or record, in the case of the Queue access method), that is accessed multiple times within a transaction only requires a single lock object for the entire transaction.

The maximum number of locks required by an application cannot be easily estimated. It is possible to calculate a maximum number of locks by multiplying the maximum number of lockers, times the maximum number of lock objects, times two (two for the two possible lock modes for each object, read and write). However, this is a pessimal value, and real applications are unlikely to actually need that many locks. Reviewing the Lock subsystem statistics is the best way to determine this value.

Standard lock modes

The Berkeley DB locking protocol is described by a conflict matrix. A conflict matrix is an NxN array in which N is the number of different lock modes supported, and the (i, j)th entry of the array indicates whether a lock of mode i conflicts with a lock of mode j. In addition, Berkeley DB defines the type **db_lockmode_t**, which is the type of a lock mode within a conflict matrix.

The following is an example of a conflict matrix. The actual conflict matrix used by Berkeley DB to support the underlying access methods is more complicated, but this matrix shows the lock mode relationships available to applications using the Berkeley DB Locking subsystem interfaces directly.

DB_LOCK_NG
: not granted (always 0)

DB_LOCK_READ
: read (shared)

DB_LOCK_WRITE
: write (exclusive)

DB_LOCK_IWRITE
: intention to write (shared)

DB_LOCK_IREAD
: intention to read (shared)

DB_LOCK_IWR
: intention to read and write (shared)

In a conflict matrix, the rows indicate the lock that is held, and the columns indicate the lock that is requested. A 1 represents a conflict (that is, do not grant the lock if the indicated lock is held), and a 0 indicates that it is OK to grant the lock.

```
  Notheld Read    Write IWrite IRead IRW
Notheld  0 0 0 0 0 0
Read*  0 0 1 1 0 1
Write**  0 1 1 1 1 1
Intent Write 0 1 1 0 0 0
Intent Read 0 0 1 0 0 0
Intent RW 0 1 1 0 0 0
```

*

In this case, suppose that there is a read lock held on an object. A new request for a read lock would be granted, but a request for a write lock would not.

**

In this case, suppose that there is a write lock held on an object. A new request for either a read or write lock would be denied.

Deadlock detection

Practically any application that uses locking may deadlock. The exceptions to this rule are when all the threads of control accessing the database are read-only or when the Berkeley DB Concurrent Data Store product is used; the Berkeley DB Concurrent Data Store product guarantees deadlock-free operation at the expense of reduced concurrency. While there are data access patterns that are deadlock free (for example, an application doing nothing but overwriting fixed-length records in an already existing database), they are extremely rare.

When a deadlock exists in the system, all the threads of control involved in the deadlock are, by definition, waiting on a lock. The deadlock detector examines the state of the lock manager and identifies a deadlock, and selects one of the lock requests to reject. (See Configuring locking (page 242) for a discussion of how a participant is selected). The DB_ENV->lock_get() or DB_ENV->lock_vec() call for which the selected participant is waiting then returns a DB_LOCK_DEADLOCK (page 230) error. When using the Berkeley DB access methods, this error return is propagated back through the Berkeley DB database handle method to the calling application.

The deadlock detector identifies deadlocks by looking for a cycle in what is commonly referred to as its "waits-for" graph. More precisely, the deadlock detector reads through the lock table, and reviews each lock object currently locked. Each object has lockers that currently hold locks on the object and possibly a list of lockers waiting for a lock on the object. Each object's list of waiting lockers defines a partial ordering. That is, for a particular object, every waiting locker comes after every holding locker because that holding locker must release its lock before the waiting locker can make forward progress. Conceptually, after each object has been examined, the partial orderings are topologically sorted. If this topological sort reveals any cycles, the lockers forming the cycle are involved in a deadlock. One of the lockers is selected for rejection.

It is possible that rejecting a single lock request involved in a deadlock is not enough to allow other lockers to make forward progress. Unfortunately, at the time a lock request is selected for rejection, there is not enough information available to determine whether rejecting that single lock request will allow forward progress or not. Because most applications have few deadlocks, Berkeley DB takes the conservative approach, rejecting as few requests as may be necessary to resolve the existing deadlocks. In particular, for each unique cycle found in the waits-for graph described in the previous paragraph, only one lock request is selected for rejection. However, if there are multiple cycles, one lock request from each cycle is selected for rejection. Only after the enclosing transactions have received the lock request rejection return and aborted their transactions can it be determined whether it is necessary to reject additional lock requests in order to allow forward progress.

The db_deadlock utility performs deadlock detection by calling the underlying Berkeley DB DB_ENV->lock_detect() method at regular intervals (DB_ENV->lock_detect() runs a single iteration of the Berkeley DB deadlock detector). Alternatively, applications can create their own deadlock utility or thread by calling the DB_ENV->lock_detect() method directly, or by using the DB_ENV->set_lk_detect() method to configure Berkeley DB to automatically run the deadlock detector whenever there is a conflict over a lock. The tradeoffs between using the DB_ENV->lock_detect() and DB_ENV->set_lk_detect() methods is that automatic deadlock detection will resolve deadlocks more quickly (because the deadlock detector runs as soon as the lock request blocks), however, automatic deadlock detection often runs the deadlock detector when there is no need for it, and for applications with large numbers of locks and/or where many operations block temporarily on locks but are soon able to proceed, automatic detection can decrease performance.

Deadlock detection using timers

Lock and transaction timeouts may be used in place of, or in addition to, regular deadlock detection. If lock timeouts are set, lock requests will return DB_LOCK_NOTGRANTED (page 230) from a lock call when it is detected that the lock's timeout has expired, that is, the lock request has blocked, waiting, longer than the specified timeout. If transaction timeouts are set, lock requests will return DB_LOCK_NOTGRANTED (page 230) from a lock call when it has been detected that the transaction has been active longer than the specified timeout.

If lock or transaction timeouts have been set, database operations will return DB_LOCK_DEADLOCK (page 230) when the lock timeout has expired or the transaction has been active longer than the specified timeout. Applications wanting to distinguish between true deadlock and timeout can use the DB_ENV->set_flags() configuration flag, which causes database operations to instead return DB_LOCK_NOTGRANTED (page 230) in the case of timeout.

As lock and transaction timeouts are only checked when lock requests first block or when deadlock detection is performed, the accuracy of the timeout depends on how often deadlock detection is performed. More specifically, transactions will continue to run after their timeout has expired if they do not block on a lock request after that time. A separate deadlock detection thread (or process) should always be used if the application depends on timeouts; otherwise, if there are no new blocked lock requests a pending timeout will never trigger.

If the database environment deadlock detector has been configured with the DB_LOCK_EXPIRE option, timeouts are the only mechanism by which deadlocks will be broken. If the deadlock detector has been configured with a different option, then regular deadlock detection will be performed, and in addition, if timeouts have also been specified, lock requests and transactions will time out as well.

Lock and transaction timeouts may be specified on a database environment wide basis using the DB_ENV->set_timeout() method. Lock timeouts may be specified on a per-lock request basis using the DB_ENV->lock_vec() method. Lock and transaction timeouts may be specified on a per-transaction basis using the DB_TXN->set_timeout() method. Per-lock and per-transaction timeouts supersede environment wide timeouts.

For example, consider that the environment wide transaction timeout has been set to 20ms, the environment wide lock timeout has been set to 10ms, a transaction has been created in

this environment and its timeout value set to 8ms, and a specific lock request has been made on behalf of this transaction where the lock timeout was set to 4ms. By default, transactions in this environment will be timed out if they block waiting for a lock after 20ms. The specific transaction described will be timed out if it blocks waiting for a lock after 8ms. By default, any lock request in this system will be timed out if it blocks longer than 10ms, and the specific lock described will be timed out if it blocks longer than 4ms.

Deadlock debugging

An occasional debugging problem in Berkeley DB applications is unresolvable deadlock. The output of the **-Co** flags of the db_stat utility can be used to detect and debug these problems. The following is a typical example of the output of this utility:

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
       1  READ        1  HELD        a.db                handle   0
80000004  WRITE       1  HELD        a.db                page     3
```

In this example, we have opened a database and stored a single key/data pair in it. Because we have a database handle open, we have a read lock on that database handle. The database handle lock is the read lock labeled *handle*. (We can normally ignore handle locks for the purposes of database debugging, as they will only conflict with other handle operations, for example, an attempt to remove the database will block because we are holding the handle locked, but reading and writing the database will not conflict with the handle lock.)

It is important to note that locker IDs are 32-bit unsigned integers, and are divided into two name spaces. Locker IDs with the high bit set (that is, values 80000000 or higher), are locker IDs associated with transactions. Locker IDs without the high bit set are locker IDs that are not associated with a transaction. Locker IDs associated with transactions map one-to-one with the transaction, that is, a transaction never has more than a single locker ID, and all of the locks acquired by the transaction will be acquired on behalf of the same locker ID.

We also hold a write lock on the database page where we stored the new key/data pair. The page lock is labeled *page* and is on page number 3. If we were to put an additional key/data pair in the database, we would see the following output:

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
80000004  WRITE       2  HELD        a.db                page     3
       1  READ        1  HELD        a.db                handle   0
```

That is, we have acquired a second reference count to page number 3, but have not acquired any new locks. If we add an entry to a different page in the database, we would acquire additional locks:

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
       1  READ        1  HELD        a.db                handle   0
80000004  WRITE       2  HELD        a.db                page     3
80000004  WRITE       1  HELD        a.db                page     2
```

Here's a simple example of one lock blocking another one:

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
80000004  WRITE       1  HELD        a.db                page     2
80000005  WRITE       1  WAIT        a.db                page     2
       1  READ        1  HELD        a.db                handle   0
80000004  READ        1  HELD        a.db                page     1
```

In this example, there are two different transactional lockers (80000004 and 80000005). Locker 80000004 is holding a write lock on page 2, and locker 80000005 is waiting for a write lock on page 2. This is not a deadlock, because locker 80000004 is not blocked on anything. Presumably, the thread of control using locker 80000004 will proceed, eventually release its write lock on page 2, at which point the thread of control using locker 80000005 can also proceed, acquiring a write lock on page 2.

If lockers 80000004 and 80000005 are not in different threads of control, the result would be *self deadlock*. Self deadlock is not a true deadlock, and won't be detected by the Berkeley DB deadlock detector. It's not a true deadlock because, if work could continue to be done on behalf of locker 80000004, then the lock would eventually be released, and locker 80000005 could acquire the lock and itself proceed. So, the key element is that the thread of control holding the lock cannot proceed because it is the same thread as is blocked waiting on the lock.

Here's an example of three transactions reaching true deadlock. First, three different threads of control opened the database, acquiring three database handle read locks.

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
       1  READ        1  HELD        a.db                handle   0
       3  READ        1  HELD        a.db                handle   0
       5  READ        1  HELD        a.db                handle   0
```

The three threads then each began a transaction, and put a key/data pair on a different page:

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
80000008  WRITE       1  HELD        a.db                page     4
       1  READ        1  HELD        a.db                handle   0
       3  READ        1  HELD        a.db                handle   0
       5  READ        1  HELD        a.db                handle   0
80000006  READ        1  HELD        a.db                page     1
80000007  READ        1  HELD        a.db                page     1
80000008  READ        1  HELD        a.db                page     1
80000006  WRITE       1  HELD        a.db                page     2
80000007  WRITE       1  HELD        a.db                page     3
```

The thread using locker 80000006 put a new key/data pair on page 2, the thread using locker 80000007, on page 3, and the thread using locker 80000008 on page 4. Because the database is a 2-level Btree, the tree was searched, and so each transaction acquired a read lock on the Btree root page (page 1) as part of this operation.

The three threads then each attempted to put a second key/data pair on a page currently locked by another thread. The thread using locker 80000006 tried to put a key/data pair on page 3, the thread using locker 80000007 on page 4, and the thread using locker 80000008 on page 2:

```
Locks grouped by object
Locker    Mode    Count   Status      ----------- Object ----------
80000008  WRITE       1  HELD        a.db                page     4
80000007  WRITE       1  WAIT        a.db                page     4
       1  READ        1  HELD        a.db                handle   0
       3  READ        1  HELD        a.db                handle   0
       5  READ        1  HELD        a.db                handle   0
80000006  READ        2  HELD        a.db                page     1
80000007  READ        2  HELD        a.db                page     1
80000008  READ        2  HELD        a.db                page     1
80000006  WRITE       1  HELD        a.db                page     2
80000008  WRITE       1  WAIT        a.db                page     2
80000007  WRITE       1  HELD        a.db                page     3
80000006  WRITE       1  WAIT        a.db                page     3
```

Now, each of the threads of control is blocked, waiting on a different thread of control. The thread using locker 80000007 is blocked by the thread using locker 80000008, due to the lock on page 4. The thread using locker 80000008 is blocked by the thread using locker 80000006, due to the lock on page 2. And the thread using locker 80000006 is blocked by the thread using locker 80000007, due to the lock on page 3. Since none of the threads of control can make progress, one of them will have to be killed in order to resolve the deadlock.

Locking granularity

With the exception of the Queue access method, the Berkeley DB access methods do page-level locking. The size of pages in a database may be set when the database is created by calling the DB->set_pagesize() method. If not specified by the application, Berkeley DB selects a page size that will provide the best I/O performance by setting the page size equal to the block size of the underlying file system. Selecting a smaller page size can result in increased concurrency for some applications.

In the Btree access method, Berkeley DB uses a technique called lock coupling to improve concurrency. The traversal of a Btree requires reading a page, searching that page to determine which page to search next, and then repeating this process on the next page. Once a page has been searched, it will never be accessed again for this operation, unless a page split is required. To improve concurrency in the tree, once the next page to read/search has been determined, that page is locked and then the original page lock is released atomically (that is, without relinquishing control of the lock manager). When page splits become necessary, write locks are reacquired.

Because the Recno access method is built upon Btree, it also uses lock coupling for read operations. However, because the Recno access method must maintain a count of records on its internal pages, it cannot lock-couple during write operations. Instead, it retains write locks on all internal pages during every update operation. For this reason, it is not possible to have high concurrency in the Recno access method in the presence of write operations.

The Queue access method uses only short-term page locks. That is, a page lock is released prior to requesting another page lock. Record locks are used for transaction isolation. The provides a high degree of concurrency for write operations. A metadata page is used to keep track of the head and tail of the queue. This page is never locked during other locking or I/O operations.

The Hash access method does not have such traversal issues, but it must always refer to its metadata while computing a hash function because it implements dynamic hashing. This metadata is stored on a special page in the hash database. This page must therefore be read-locked on every operation. Fortunately, it needs to be write-locked only when new pages are allocated to the file, which happens in three cases:

- a hash bucket becomes full and needs to split
- a key or data item is too large to fit on a normal page
- the number of duplicate items for a fixed key becomes so large that they are moved to an auxiliary page

In this case, the access method must obtain a write lock on the metadata page, thus requiring that all readers be blocked from entering the tree until the update completes.

Finally, when traversing duplicate data items for a key, the lock on the key value also acts as a lock on all duplicates of that key. Therefore, two conflicting threads of control cannot access the same duplicate set simultaneously.

Locking without transactions

If an application runs with locking specified, but not transactions (for example, DB_ENV->open() is callsed with DB_INIT_LOCK or DB_INIT_CDB specified, but not DB_INIT_TXN), locks are normally acquired during each Berkeley DB operation and released before the operation returns to the caller. The only exception is in the case of cursor operations. Cursors identify a particular position in a file. For this reason, cursors must retain read locks across cursor calls to make sure that the position is uniquely identifiable during a subsequent cursor call, and so that an operation using DB_CURRENT will always refer to the same record as a previous cursor call. These cursor locks cannot be released until the cursor is either repositioned and a new cursor lock established (for example, using the DB_NEXT or DB_SET flags), or the cursor is closed. As a result, application writers are encouraged to close cursors as soon as possible.

It is important to realize that concurrent applications that use locking must ensure that two concurrent threads do not block each other. However, because Btree and Hash access method page splits can occur at any time, there is virtually no way to guarantee that an application that writes the database cannot deadlock. Applications running without the protection of transactions may deadlock, and can leave the database in an inconsistent state when they do so. Applications that need concurrent access, but not transactions, are more safely implemented using the Berkeley DB Concurrent Data Store Product.

Locking with transactions: two-phase locking

Berkeley DB uses a locking protocol called *two-phase locking (2PL)*. This is the traditional protocol used in conjunction with lock-based transaction systems.

In a two-phase locking system, transactions are divided into two distinct phases. During the first phase, the transaction only acquires locks; during the second phase, the transaction only releases locks. More formally, once a transaction releases a lock, it may not acquire any additional locks. Practically, this translates into a system in which locks are acquired as they are needed throughout a transaction and retained until the transaction ends, either by committing or aborting. In Berkeley DB, locks are released during DB_TXN->abort() or DB_TXN->commit(). The only exception to this protocol occurs when we use lock-coupling to traverse a data structure. If the locks are held only for traversal purposes, it is safe to release locks before transactions commit or abort.

For applications, the implications of 2PL are that long-running transactions will hold locks for a long time. When designing applications, lock contention should be considered. In order to reduce the probability of deadlock and achieve the best level of concurrency possible, the following guidelines are helpful.

1. When accessing multiple databases, design all transactions so that they access the files in the same order.

2. If possible, access your most hotly contested resources last (so that their locks are held for the shortest time possible).

3. If possible, use nested transactions to protect the parts of your transaction most likely to deadlock.

Berkeley DB Concurrent Data Store locking conventions

The Berkeley DB Concurrent Data Store product has a simple set of conventions for locking. It provides multiple-reader/single-writer semantics, but not per-page locking or transaction recoverability. As such, it does its locking entirely in the Berkeley DB interface layer.

The object it locks is the file, identified by its unique file number. The locking matrix is not one of the two standard lock modes, instead, we use a four-lock set, consisting of the following:

DB_LOCK_NG
: not granted (always 0)

DB_LOCK_READ
: read (shared)

DB_LOCK_WRITE
: write (exclusive)

DB_LOCK_IWRITE
: intention-to-write (shared with NG and READ, but conflicts with WRITE and IWRITE)

The IWRITE lock is used for cursors that will be used for updating (IWRITE locks are implicitly obtained for write operations through the Berkeley DB handles, for example, DB->put() or DB->del()). While the cursor is reading, the IWRITE lock is held; but as soon as the cursor is about to modify the database, the IWRITE is upgraded to a WRITE lock. This upgrade blocks until all readers have exited the database. Because only one IWRITE lock is allowed at any one time, no two cursors can ever try to upgrade to a WRITE lock at the same time, and therefore deadlocks are prevented, which is essential because Berkeley DB Concurrent Data Store does not include deadlock detection and recovery.

Applications that need to lock compatibly with Berkeley DB Concurrent Data Store must obey the following rules:

1. Use only lock modes DB_LOCK_NG, DB_LOCK_READ, DB_LOCK_WRITE, DB_LOCK_IWRITE.

2. Never attempt to acquire a WRITE lock on an object that is already locked with a READ lock.

Berkeley DB Transactional Data Store locking conventions

All Berkeley DB access methods follow the same conventions for locking database objects. Applications that do their own locking and also do locking via the access methods must be careful to adhere to these conventions.

Whenever a Berkeley DB database is opened, the DB handle is assigned a unique locker ID. Unless transactions are specified, that ID is used as the locker for all calls that the Berkeley DB methods make to the lock subsystem. In order to lock a file, pages in the file, or records in the file, we must create a unique ID that can be used as the object to be locked in calls to the lock manager. Under normal operation, that object is a 28-byte value created by the concatenation of a unique file identifier, a page or record number, and an object type (page or record).

In a transaction-protected environment, database create and delete operations are recoverable and single-threaded. This single-threading is achieved using a single lock for the entire environment that must be acquired before beginning a create or delete operation. In this case, the object on which Berkeley DB will lock is a 4-byte unsigned integer with a value of 0.

If applications are using the lock subsystem directly while they are also using locking via the access methods, they must take care not to inadvertently lock objects that happen to be equal to the unique file IDs used to lock files. This is most easily accomplished by using a lock object with a length different from the values used by Berkeley DB.

All the access methods other than Queue use standard read/write locks in a simple multiple-reader/single writer page-locking scheme. An operation that returns data (for example, DB->get() or DBC->get()) obtains a read lock on all the pages accessed while locating the requested record. When an update operation is requested (for example, DB->put() or DBC->del()), the page containing the updated (or new) data is write-locked. As read-modify-write cycles are quite common and are deadlock-prone under normal circumstances, the Berkeley DB interfaces allow the application to specify the DB_RMW flag, which causes operations to immediately obtain a write lock, even though they are only reading the data. Although this may reduce concurrency somewhat, it reduces the probability of deadlock. In the presence of transactions, page locks are held until transaction commit.

The Queue access method does not hold long-term page locks. Instead, page locks are held only long enough to locate records or to change metadata on a page, and record locks are held for the appropriate duration. In the presence of transactions, record locks are held until transaction commit. For Berkeley DB operations, record locks are held until operation completion; for DBC operations, record locks are held until subsequent records are returned or the cursor is closed.

Under non-transaction operations, the access methods do not normally hold locks across calls to the Berkeley DB interfaces. The one exception to this rule is when cursors are used. Because cursors maintain a position in a file, they must hold locks across calls; in fact, they will hold a lock until the cursor is closed.

In this mode, the assignment of locker IDs to DB and cursor handles is complicated. If the DB_THREAD option was specified when the DB handle was opened, each use of DB has its own unique locker ID, and each cursor is assigned its own unique locker ID when it is created, so DB handle and cursor operations can all conflict with one another. (This is because when Berkeley DB handles may be shared by multiple threads of control the Berkeley DB library cannot identify which operations are performed by which threads of control, and it must ensure that two different threads of control are not simultaneously modifying the same data structure. By assigning each DB handle and cursor its own locker, two threads of control sharing a handle cannot inadvertently interfere with each other.)

This has important implications. If a single thread of control opens two cursors, uses a combination of cursor and non-cursor operations, or begins two separate transactions, the operations are performed on behalf of different lockers. Conflicts that arise between these different lockers may not cause actual deadlocks, but can, in fact, permanently block the thread of control. For example, assume that an application creates a cursor and uses it to read record A. Now, assume a second cursor is opened, and the application attempts to write record A using the second cursor. Unfortunately, the first cursor has a read lock, so the second cursor cannot obtain its write lock. However, that read lock is held by the same thread of control, so the read lock can never be released if we block waiting for the write lock. This might appear to be a deadlock from the application's perspective, but Berkeley DB cannot identify it as such because it has no knowledge of which lockers belong to which threads of control. For this reason, application designers are encouraged to close cursors as soon as they are done with them.

If the DB_THREAD option was not specified when the DB handle was opened, all uses of the DB handle and all cursors created using that handle will use the same locker ID for all operations. In this case, if a single thread of control opens two cursors or uses a combination of cursor and non-cursor operations, these operations are performed on behalf of the same locker, and so cannot deadlock or block the thread of control.

Complicated operations that require multiple cursors (or combinations of cursor and non-cursor operations) can be performed in two ways. First, they may be performed within a transaction, in which case all operations lock on behalf of the designated transaction. Second, they may be performed using a local DB handle, although, as DB->open() operations are relatively slow, this may not be a good idea. Finally, the DBC->dup() function duplicates a cursor, using the same locker ID as the originating cursor. There is no way to achieve this duplication functionality through the DB handle calls, but any DB call can be implemented by one or more calls through a cursor.

When the access methods use transactions, many of these problems disappear. The transaction ID is used as the locker ID for all operations performed on behalf of the transaction. This means that the application may open multiple cursors on behalf of the same transaction and these cursors will all share a common locker ID. This is safe because transactions cannot span threads of control, so the library knows that two cursors in the same transaction cannot modify the database concurrently.

Locking and non-Berkeley DB applications

The Lock subsystem is useful outside the context of Berkeley DB. It can be used to manage concurrent access to any collection of either ephemeral or persistent objects. That is, the lock region can persist across invocations of an application, so it can be used to provide long-term locking (for example, conference room scheduling).

In order to use the locking subsystem in such a general way, the applications must adhere to a convention for identifying objects and lockers. Consider a conference room scheduling problem, in which there are three conference rooms scheduled in half-hour intervals. The scheduling application must then select a way to identify each conference room/time slot combination. In this case, we could describe the objects being locked as bytestrings consisting of the conference room name, the date when it is needed, and the beginning of the appropriate half-hour slot.

Lockers are 32-bit numbers, so we might choose to use the User ID of the individual running the scheduling program. To schedule half-hour slots, all the application needs to do is issue a DB_ENV->lock_get() call for the appropriate locker/object pair. To schedule a longer slot, the application needs to issue a DB_ENV->lock_vec() call, with one DB_ENV->lock_get() operation per half-hour — up to the total length. If the DB_ENV->lock_vec() call fails, the application would have to release the parts of the time slot that were obtained.

To cancel a reservation, the application would make the appropriate DB_ENV->lock_put() calls. To reschedule a reservation, the DB_ENV->lock_get() and DB_ENV->lock_put() calls could all be made inside of a single DB_ENV->lock_vec() call. The output of DB_ENV->lock_stat() could be post-processed into a human-readable schedule of conference room use.

Chapter 16. The Logging Subsystem

Introduction to the logging subsystem

The Logging subsystem is the logging facility used by Berkeley DB. It is largely Berkeley DB-specific, although it is potentially useful outside of the Berkeley DB package for applications wanting write-ahead logging support. Applications wanting to use the log for purposes other than logging file modifications based on a set of open file descriptors will almost certainly need to make source code modifications to the Berkeley DB code base.

A log can be shared by any number of threads of control. The DB_ENV->open() method is used to open a log. When the log is no longer in use, it should be closed using the DB_ENV->close() method.

Individual log entries are identified by log sequence numbers. Log sequence numbers are stored in an opaque object, an DB_LSN.

The DB_ENV->log_cursor() method is used to allocate a log cursor. Log cursors have two methods: DB_LOGC->get() method to retrieve log records from the log, and DB_LOGC->close() method to destroy the cursor.

There are additional methods for integrating the log subsystem with a transaction processing system:

DB_ENV->log_flush()

Flushes the log up to a particular log sequence number.

DB_ENV->log_compare()

Allows applications to compare any two log sequence numbers.

DB_ENV->log_file()

Maps a log sequence number to the specific log file that contains it.

DB_ENV->log_archive()

Returns various sets of log filenames. These methods are used for database administration; for example, to determine if log files may safely be removed from the system.

DB_ENV->log_stat()

The display db_stat utility utility used the DB_ENV->log_stat() method to display statistics about the log.

DB_ENV->remove()

The log meta-information (but not the log files themselves) may be removed using the DB_ENV->remove() method.

Logging Subsystem and Related Methods	Description
DB_LSN	Log Sequence Numbers
DB_ENV->log_compare()	Compare two Log Sequence Numbers

Logging Subsystem and Related Methods	Description
DB_ENV->log_archive()	List log and database files
DB_ENV->log_file()	Map Log Sequence Numbers to log files
DB_ENV->log_flush()	Flush log records
DB_ENV->log_printf()	Append informational message to the log
DB_ENV->log_put()	Write a log record
DB_ENV->log_stat()	Return log subsystem statistics
Logging Subsystem Cursors	
DB_ENV->log_cursor()	Create a log cursor handle
DB_LOGC->close()	Close a log cursor
DB_LOGC->get()	Retrieve a log record
Logging Subsystem Configuration	
DB_ENV->log_set_config()	Configure the logging subsystem
DB_ENV->set_lg_bsize()	Set log buffer size
DB_ENV->set_lg_dir()	Set the environment logging directory
DB_ENV->set_lg_filemode()	Set log file mode
DB_ENV->set_lg_max()	Set log file size
DB_ENV->set_lg_regionmax()	Set logging region size

Configuring logging

The aspects of logging that may be configured are the size of the logging subsystem's region, the size of the log files on disk and the size of the log buffer in memory. The DB_ENV->set_lg_regionmax() method specifies the size of the logging subsystem's region, in bytes. The logging subsystem's default size is approximately 60KB. This value may need to be increased if a large number of files are registered with the Berkeley DB log manager, for example, by opening a large number of Berkeley DB database files in a transactional application.

The DB_ENV->set_lg_max() method specifies the individual log file size for all the applications sharing the Berkeley DB environment. Setting the log file size is largely a matter of convenience and a reflection of the application's preferences in backup media and frequency. However, setting the log file size too low can potentially cause problems because it would be possible to run out of log sequence numbers, which requires a full archival and application restart to reset. See Log file limits (page 258) for more information.

The DB_ENV->set_lg_bsize() method specifies the size of the in-memory log buffer, in bytes. Log information is stored in memory until the buffer fills up or transaction commit forces the buffer to be written to disk. Larger buffer sizes can significantly increase throughput in the presence of long-running transactions, highly concurrent applications, or transactions producing large amounts of data. By default, the buffer is approximately 32KB.

The DB_ENV->set_lg_dir() method specifies the directory in which log files will be placed. By default, log files are placed in the environment home directory.

The DB_ENV->set_lg_filemode() method specifies the absolute file mode for created log files. This method is only useful for the rare Berkeley DB application that does not control its umask value.

The DB_ENV->log_set_config() method configures several boolean parameters that control the use of file system controls such as O_DIRECT and O_DSYNC, automatic removal of log files, in-memory logging, and pre-zeroing of logfiles.

Log file limits

Log filenames and sizes impose a limit on how long databases may be used in a Berkeley DB database environment. It is quite unlikely that an application will reach this limit; however, if the limit is reached, the Berkeley DB environment's databases must be dumped and reloaded.

The log filename consists of **log.** followed by 10 digits, with a maximum of 2,000,000,000 log files. Consider an application performing 6000 transactions per second for 24 hours a day, logged into 10MB log files, in which each transaction is logging approximately 500 bytes of data. The following calculation:

```
(10 * 2^20 * 2000000000) / (6000 * 500 * 365 * 60 * 60 * 24) = ~221
```

indicates that the system will run out of log filenames in roughly 221 years.

There is no way to reset the log filename space in Berkeley DB. If your application is reaching the end of its log filename space, you must do the following:

1. Archive your databases as if to prepare for catastrophic failure (see Database and log file archival (page 173) for more information).

2. Reset the database's log sequence numbers (see the **-r** option to the db_load utility for more information).

3. Remove all of the log files from the database environment. (This is the only situation in which all the log files are removed from an environment; in all other cases, at least a single log file is retained.)

4. Restart your application.

Chapter 17. The Memory Pool Subsystem

Introduction to the memory pool subsystem

The Memory Pool subsystem is the general-purpose shared memory buffer pool used by Berkeley DB. This module is useful outside of the Berkeley DB package for processes that require page-oriented, shared and cached file access. (However, such "use outside of Berkeley DB" is not supported in replicated environments.)

A *memory pool* is a memory cache shared among any number of threads of control. The DB_INIT_MPOOL flag to the DB_ENV->open() method opens and optionally creates a memory pool. When that pool is no longer in use, it should be closed using the DB_ENV->close() method.

The DB_ENV->memp_fcreate() method returns a DB_MPOOLFILE handle on an underlying file within the memory pool. The file may be opened using the DB_MPOOLFILE->open() method. The DB_MPOOLFILE->get() method is used to retrieve pages from files in the pool. All retrieved pages must be subsequently returned using the DB_MPOOLFILE->put() method. At the time pages are returned, they may be marked **dirty**, which causes them to be written to the underlying file before being discarded from the pool. If there is insufficient room to bring a new page in the pool, a page is selected to be discarded from the pool using a least-recently-used algorithm. All dirty pages in the pool from the file may be flushed using the DB_MPOOLFILE->sync() method. When the file handle is no longer in use, it should be closed using the DB_MPOOLFILE->close() method.

There are additional configuration interfaces that apply when opening a new file in the memory pool:

- The DB_MPOOLFILE->set_clear_len() method specifies the number of bytes to clear when creating a new page in the memory pool.
- The DB_MPOOLFILE->set_fileid() method specifies a unique ID associated with the file.
- The DB_MPOOLFILE->set_ftype() method specifies the type of file for the purposes of page input and output processing.
- The DB_MPOOLFILE->set_lsn_offset() method specifies the byte offset of each page's log sequence number (DB_LSN) for the purposes of transaction checkpoints.
- The DB_MPOOLFILE->set_pgcookie() method specifies an application provided argument for the purposes of page input and output processing.

There are additional interfaces for the memory pool as a whole:

- It is possible to gradually flush buffers from the pool in order to maintain a consistent percentage of clean buffers in the pool using the DB_ENV->memp_trickle() method.
- Because special-purpose processing may be necessary when pages are read or written (for example, endian conversion, or page checksums), the DB_ENV->memp_register() function allows applications to specify automatic input and output processing in these cases.

- The db_stat utility uses the DB_ENV->memp_stat() method to display statistics about the efficiency of the pool.
- All dirty pages in the pool may be flushed using the DB_ENV->memp_sync() method. In addition, DB_ENV->memp_sync() takes an argument that is specific to database systems, and which allows the memory pool to be flushed up to a specified log sequence number (DB_LSN).
- The entire pool may be discarded using the DB_ENV->remove() method.

Memory Pools and Related Methods	Description
DB->get_mpf() handle	Return DB's underlying DB_MPOOLFILE handle
DB_ENV->memp_stat()	Return memory pool statistics
DB_ENV->memp_sync()	Flush pages from a memory pool
DB_ENV->memp_trickle()	Trickle flush pages from a memory pool
Memory Pool Configuration	
DB_ENV->memp_register()	Register input/output functions for a file in a memory pool
DB_ENV->set_cachesize()	Set the environment cache size
DB_ENV->set_cache_max()	Set the maximum cache size
DB_ENV->set_mp_max_openfd()	Set the maximum number of open file descriptors
DB_ENV->set_mp_max_write()	Set the maximum number of sequential disk writes
DB_ENV->set_mp_mmapsize()	Set maximum mapped-in database file size
Memory Pool Files	
DB_ENV->memp_fcreate()	Create a memory pool file handle
DB_MPOOLFILE->close()	Close a file in a memory pool
DB_MPOOLFILE->get()	Get page from a file in a memory pool
DB_MPOOLFILE->open()	Open a file in a memory pool
DB_MPOOLFILE->put()	Return a page to a memory pool
DB_MPOOLFILE->sync()	Flush pages from a file in a memory pool
Memory Pool File Configuration	
DB_MPOOLFILE->set_clear_len()	Set file page bytes to be cleared
DB_MPOOLFILE->set_fileid()	Set file unique identifier
DB_MPOOLFILE->set_flags()	General memory pool file configuration
DB_MPOOLFILE->set_ftype()	Set file type
DB_MPOOLFILE->set_lsn_offset()	Set file log-sequence-number offset
DB_MPOOLFILE->set_maxsize()	Set maximum file size

Memory Pools and Related Methods	Description
DB_MPOOLFILE->set_pgcookie()	Set file cookie for pgin/pgout
mempset_priority()	Set memory pool file priority

Configuring the memory pool

There are two issues to consider when configuring the memory pool.

The first issue, the most important tuning parameter for Berkeley DB applications, is the size of the memory pool. There are two ways to specify the pool size. First, calling the DB_ENV->set_cachesize() method specifies the pool size for all of the applications sharing the Berkeley DB environment. Second, the DB->set_cachesize() method only specifies a pool size for the specific database. Note: It is meaningless to call DB->set_cachesize() for a database opened inside of a Berkeley DB environment because the environment pool size will override any pool size specified for a single database. For information on tuning the Berkeley DB cache size, see Selecting a cache size (page 20).

The second memory pool configuration issue is the maximum size an underlying file can be and still be mapped into the process address space (instead of reading the file's pages into the cache). Mapping files into the process address space can result in better performance because available virtual memory is often much larger than the local cache, and page faults are faster than page copying on many systems. However, in the presence of limited virtual memory, it can cause resource starvation; and in the presence of large databases, it can result in immense process sizes. In addition, because of the requirements of the Berkeley DB transactional implementation, only read-only files can be mapped into process memory.

To specify that no files are to be mapped into the process address space, specify the DB_NOMMAP flag to the DB_ENV->set_flags() method. To specify that any individual file should not be mapped into the process address space, specify the DB_NOMMAP flag to the DB_MPOOLFILE->open() interface. To limit the size of files mapped into the process address space, use the DB_ENV->set_mp_mmapsize() method.

Chapter 18. The Transaction Subsystem

Introduction to the transaction subsystem

The Transaction subsystem makes operations atomic, consistent, isolated, and durable in the face of system and application failures. The subsystem requires that the data be properly logged and locked in order to attain these properties. Berkeley DB contains all the components necessary to transaction-protect the Berkeley DB access methods, and other forms of data may be protected if they are logged and locked appropriately.

The Transaction subsystem is created, initialized, and opened by calls to DB_ENV->open() with the DB_INIT_TXN flag specified. Note that enabling transactions automatically enables logging, but does not enable locking because a single thread of control that needed atomicity and recoverability would not require it.

The DB_ENV->txn_begin() function starts a transaction, returning an opaque handle to a transaction. If the parent parameter to DB_ENV->txn_begin() is non-NULL, the new transaction is a child of the designated parent transaction.

The DB_TXN->abort() function ends the designated transaction and causes all updates performed by the transaction to be undone. The end result is that the database is left in a state identical to the state that existed prior to the DB_ENV->txn_begin(). If the aborting transaction has any child transactions associated with it (even ones that have already been committed), they are also aborted. Any transactions that are unresolved (neither committed nor aborted) when the application or system fails are aborted during recovery.

The DB_TXN->commit() function ends the designated transaction and makes all the updates performed by the transaction permanent, even in the face of application or system failure. If this is a parent transaction committing, all child transactions that individually committed or had not been resolved are also committed.

Transactions are identified by 32-bit unsigned integers. The ID associated with any transaction can be obtained using the DB_TXN->id() function. If an application is maintaining information outside of Berkeley DB it wants to transaction-protect, it should use this transaction ID as the locking ID.

The DB_ENV->txn_checkpoint() function causes a transaction checkpoint. A checkpoint is performed using to a specific log sequence number (LSN), referred to as the checkpoint LSN. When a checkpoint completes successfully, it means that all data buffers whose updates are described by LSNs less than the checkpoint LSN have been written to disk. This, in turn, means that the log records less than the checkpoint LSN are no longer necessary for normal recovery (although they would be required for catastrophic recovery if the database files were lost), and all log files containing only records prior to the checkpoint LSN may be safely archived and removed.

The time required to run normal recovery is proportional to the amount of work done between checkpoints. If a large number of modifications happen between checkpoints, many updates recorded in the log may not have been written to disk when failure occurred, and recovery may take longer to run. Generally, if the interval between checkpoints is short, data may be

being written to disk more frequently, but the recovery time will be shorter. Often, the checkpoint interval is tuned for each specific application.

The DB_TXN->stat() method returns information about the status of the transaction subsystem. It is the programmatic interface used by the db_stat utility.

The transaction system is closed by a call to DB_ENV->close().

Finally, the entire transaction system may be removed using the DB_ENV->remove() method.

Transaction Subsystem and Related Methods	Description
DB_ENV->txn_checkpoint()	Checkpoint the transaction subsystem
DB_TXN->recover()	Distributed transaction recovery
DB_TXN->stat()	Return transaction subsystem statistics
DB_ENV->set_timeout()	Set lock and transaction timeout
DB_ENV->set_tx_max()	Set maximum number of transactions
DB_ENV->set_tx_timestamp()	Set recovery timestamp
DB_ENV->txn_begin()	Begin a transaction
DB_TXN->abort()	Abort a transaction
DB_TXN->commit()	Commit a transaction
DB_TXN->discard()	Discard a prepared but not resolved transaction handle
DB_TXN->id()	Return a transaction's ID
DB_TXN->prepare()	Prepare a transaction for commit
DB_TXN->set_name()	Associate a string with a transaction
DB_TXN->set_timeout()	Set transaction timeout

Configuring transactions

The application may change the number of simultaneous outstanding transactions supported by the Berkeley DB environment by calling the DB_ENV->set_tx_max() method. This will also set the size of the underlying transaction subsystem's region. When the number of outstanding transactions is reached, additional calls to DB_ENV->txn_begin() will fail until some active transactions complete.

The application can limit how long a transaction runs or blocks on contested resources. The DB_ENV->set_timeout() method specifies the length of the timeout. This value is checked whenever deadlock detection is performed or when the transaction is about to block on a lock that cannot be immediately granted. Because timeouts are only checked at these times, the accuracy of the timeout depends on how often deadlock detection is performed or how frequently the transaction blocks.

There is an additional parameter used in configuring transactions; the DB_TXN_NOSYNC. Setting the DB_TXN_NOSYNC flag to DB_ENV->set_flags() when opening a transaction region changes

the behavior of transactions to not write or synchronously flush the log during transaction commit.

This change may significantly increase application transactional throughput. However, it means that although transactions will continue to exhibit the ACI (atomicity, consistency, and isolation) properties, they will not have D (durability). Database integrity will be maintained, but it is possible that some number of the most recently committed transactions may be undone during recovery instead of being redone.

Transaction limits

Transaction IDs

Transactions are identified by 31-bit unsigned integers, which means there are just over two billion unique transaction IDs. When a database environment is initially created or recovery is run, the transaction ID name space is reset, and new transactions are numbered starting from 0x80000000 (2,147,483,648). The IDs will wrap if the maximum transaction ID is reached, starting again from 0x80000000. The most recently allocated transaction ID is the **st_last_txnid** value in the transaction statistics information, and can be displayed by the db_stat utility.

Cursors

When using transactions, cursors are localized to a single transaction. That is, a cursor may not span transactions, and must be opened and closed within a single transaction. In addition, intermingling transaction-protected cursor operations and non-transaction-protected cursor operations on the same database in a single thread of control is practically guaranteed to deadlock because the locks obtained for transactions and non-transactions can conflict.

Multiple Threads of Control

Because transactions must hold all their locks until commit, a single transaction may accumulate a large number of long-term locks during its lifetime. As a result, when two concurrently running transactions access the same database, there is strong potential for conflict. Although Berkeley DB allows an application to have multiple outstanding transactions active within a single thread of control, great care must be taken to ensure that the transactions do not block each other (for example, attempt to obtain conflicting locks on the same data). If two concurrently active transactions in the same thread of control do encounter a lock conflict, the thread of control will deadlock so that the deadlock detector cannot detect the problem. In this case, there is no true deadlock, but because the transaction on which a transaction is waiting is in the same thread of control, no forward progress can be made.

Chapter 19. Sequences

Introduction to sequences

Sequences provide an arbitrary number of persistent objects that return an increasing or decreasing sequence of integers. Opening a sequence handle associates it with a record in a database. The handle can maintain a cache of values from the database so that a database update is not needed as the application allocates a value.

A sequence is stored as a record pair in a database. The database may be of any type, but may not have been configured to support duplicate data items. The sequence is referenced by the key used when the sequence is created, therefore the key must be compatible with the underlying access method. If the database stores fixed-length records, the record size must be at least 64 bytes long.

Since a sequence handle is opened using a database handle, the use of transactions with the sequence must follow how the database handle was opened. In other words, if the database handle was opened within a transaction, operations on the sequence handle must use transactions. Of course, if sequences are cached, not all operations will actually trigger a transaction.

For the highest concurrency, caching should be used and the DB_AUTO_COMMIT and DB_TXN_NOSYNC flags should be specified to the DB_SEQUENCE->get() method call. If the allocation of the sequence value must be part of a transaction, and rolled back if the transaction aborts, then no caching should be specified and the transaction handle must be passed to the DB_SEQUENCE->get() method.

Sequences and Related Methods	Description
Sequence class	Create a sequence handle
DB_SEQUENCE->close()	Close a sequence
DB_SEQUENCE->get()	Get the next sequence element(s)
DB_SEQUENCE->open()	Return a handle for the underlying sequence database
DB_SEQUENCE->open()	Return the key for a sequence
DB_SEQUENCE->initial_value()	Set the initial value of a sequence
DB_SEQUENCE->open()	Open a sequence
DB_SEQUENCE->remove()	Remove a sequence
DB_SEQUENCE->stat()	Return sequence statistics
Sequences Configuration	
DB_SEQUENCE->set_cachesize()	Set the cache size of a sequence
DB_SEQUENCE->set_flags()	Set the flags for a sequence
DB_SEQUENCE->set_range()	Set the range for a sequence

Chapter 20. Berkeley DB Extensions: Tcl

Loading Berkeley DB with Tcl

Berkeley DB includes a dynamically loadable Tcl API, which requires that Tcl/Tk 8.4 or later already be installed on your system. You can download a copy of Tcl from the Tcl Developer Xchange [http://www.tcl.tk] Web site.

This document assumes that you already configured Berkeley DB for Tcl support, and you have built and installed everything where you want it to be. If you have not done so, see Configuring Berkeley DB (page 290) or Building the Tcl API (page 315) for more information.

Installing as a Tcl Package

Once enabled, the Berkeley DB shared library for Tcl is automatically installed as part of the standard installation process. However, if you want to be able to dynamically load it as a Tcl package into your script, there are several steps that must be performed:

1. Run the Tcl shell in the install directory.

2. Append this directory to your auto_path variable.

3. Run the pkg_mkIndex proc, giving the name of the Berkeley DB Tcl library.

For example:

```
# tclsh8.4
% lappend auto_path /usr/local/BerkeleyDB.4.8/lib
% pkg_mkIndex /usr/local/BerkeleyDB.4.8/lib libdb_tcl-4.8.so
```

Note that your Tcl and Berkeley DB version numbers may differ from the example, and so your tclsh and library names may be different.

Loading Berkeley DB with Tcl

The Berkeley DB package may be loaded into the user's interactive Tcl script (or wish session) via the **load** command. For example:

```
load /usr/local/BerkeleyDB.4.8/lib/libdb_tcl-4.8.so
```

Note that your Berkeley DB version numbers may differ from the example, and so the library name may be different.

If you installed your library to run as a Tcl package, Tcl application scripts should use the **package** command to indicate to the Tcl interpreter that it needs the Berkeley DB package and where to find it. For example:

```
lappend auto_path "/usr/local/BerkeleyDB.4.8/lib"
package require Db_tcl
```

No matter which way the library gets loaded, it creates a command named **berkdb**. All the Berkeley DB functionality is accessed via this command and additional commands it creates on behalf of the application. A simple test to determine whether everything is loaded and ready is to display the library version, as follows:

```
berkdb version -string
```

This should return you the Berkeley DB version in a string format.

Using Berkeley DB with Tcl

All commands in the Berkeley DB Tcl interface are in the following form:

```
command_handle operation options
```

The *command handle* is **berkdb** or one of the additional commands that may be created. The *operation* is what you want to do to that handle, and the *options* apply to the operation. Commands that get created on behalf of the application have their own sets of operations. Generally, any calls in DB that result in new object handles will translate into a new command handle in Tcl. Then, the user can access the operations of the handle via the new Tcl command handle.

Newly created commands are named with an abbreviated form of their objects, followed by a number. Some created commands are subcommands of other created commands and will be the first command, followed by a period (.), and then followed by the new subcommand. For example, suppose that you have a database already existing called my_data.db. The following example shows the commands created when you open the database and when you open a cursor:

```
# First open the database and get a database command handle
% berkdb open my_data.db
db0
#Get some data from that database
% db0 get my_key
{{my_key my_data0}{my_key my_data1}}
#Open a cursor in this database, get a new cursor handle
% db0 cursor
db0.c0
#Get the first data from the cursor
% db0.c0 get -first
{{first_key first_data}}
```

All commands in the library support a special option **-?** that will list the correct operations for a command or the correct options.

A list of commands and operations can be found in the Tcl API documentation.

Tcl API programming notes

The Berkeley DB Tcl API does not attempt to avoid evaluating input as Tcl commands. For this reason, it may be dangerous to pass unreviewed user input through the Berkeley DB Tcl API,

as the input may subsequently be evaluated as a Tcl command. Additionally, the Berkeley DB Tcl API initialization routine resets process' effective user and group IDs to the real user and group IDs, to minimize the effectiveness of a Tcl injection attack.

The Tcl API closely parallels the Berkeley DB programmatic interfaces. If you are already familiar with one of those interfaces, there will not be many surprises in the Tcl API.

The Tcl API currently does not support multithreading although it could be made to do so. The Tcl shell itself is not multithreaded and the Berkeley DB extensions use global data unprotected from multiple threads.

Several pieces of Berkeley DB functionality are not available in the Tcl API. Any of the functions that require a user-provided function are not supported via the Tcl API. For example, there is no equivalent to the DB->set_dup_compare() or DB_ENV->set_errcall() methods. methods.

Tcl error handling

The Tcl interfaces to Berkeley DB generally return TCL_OK on success and throw a Tcl error on failure, using the appropriate Tcl interfaces to provide the user with an informative error message. There are some "expected" failures, however, for which no Tcl error will be thrown and for which Tcl commands will return TCL_OK. These failures include times when a searched-for key is not found, a requested key/data pair was previously deleted, or a key/data pair cannot be written because the key already exists.

These failures can be detected by searching the Berkeley DB error message that is returned. For example, use the following to detect that an attempt to put a record into the database failed because the key already existed:

```
% berkdb open -create -btree a.db
db0
% db0 put dog cat
0
% set ret [db0 put -nooverwrite dog newcat]
DB_KEYEXIST: Key/data pair already exists
% if { [string first DB_KEYEXIST $ret] != -1 } {
 puts "This was an error; the key existed"
}
This was an error; the key existed
% db0 close
0
% exit
```

To simplify parsing, it is recommended that the initial Berkeley DB error name be checked; for example, DB_MULTIPLE in the previous example. To ensure that Tcl scripts are not broken by upgrading to new releases of Berkeley DB, these values will not change in future releases of Berkeley DB. There are currently only three such "expected" error returns:

```
DB_NOTFOUND: No matching key/data pair found
DB_KEYEMPTY: Nonexistent key/data pair
DB_KEYEXIST: Key/data pair already exists
```

Finally, sometimes Berkeley DB will output additional error information when a Berkeley DB error occurs. By default, all Berkeley DB error messages will be prefixed with the created command in whose context the error occurred (for example, "env0", "db2", and so on). There are several ways to capture and access this information.

First, if Berkeley DB invokes the error callback function, the additional information will be placed in the error result returned from the command and in the errorInfo backtrace variable in Tcl.

Also, the two calls to open an environment and open a database take an option, **-errfile filename**, which sets an output file to which these additional error messages should be written.

Additionally, the two calls to open an environment and open a database take an option, **-errpfx string**, which sets the error prefix to the given string. This option may be useful in circumstances where a more descriptive prefix is desired or where a constant prefix indicating an error is desired.

Tcl FAQ

1. **I have several versions of Tcl installed. How do I configure Berkeley DB to use a particular version?**

   To compile the Tcl interface with a particular version of Tcl, use the --with-tcl option to specify the Tcl installation directory that contains the tclConfig.sh file. See Changing compile or load options (page 295) for more information.

2. **Berkeley DB was configured using --enable-tcl or --with-tcl and fails to build.**

   The Berkeley DB Tcl interface requires Tcl version 8.4 or greater.

3. **Berkeley DB was configured using --enable-tcl or --with-tcl and fails to build.**

   If the Tcl installation was moved after it was configured and installed, try reconfiguring and reinstalling Tcl.

   Also, some systems do not search for shared libraries by default, or do not search for shared libraries named the way the Tcl installation names them, or are searching for a different kind of library than those in your Tcl installation. For example, Linux systems often require linking "libtcl.a" to "libtcl#.#.a", whereas AIX systems often require adding the "-brtl" flag to the linker. A simpler solution that almost always works on all systems is to create a link from "libtcl.#.#.a" or "libtcl.so" (or whatever you happen to have) to "libtcl.a" and reconfigure.

4. **Loading the Berkeley DB library into Tcl on AIX causes a core dump.**

   In some versions of Tcl, the "tclConfig.sh" autoconfiguration script created by the Tcl installation does not work properly under AIX, and you may have to modify values in the tclConfig.sh file to in order to load the Berkeley DB library into Tcl. Specifically, the TCL_LIB_SPEC variable should contain sufficient linker flags to find and link against the installed libtcl library. In some circumstances, the tclConfig.sh file built by Tcl does not.

Chapter 21. Berkeley DB Extensions

Using Berkeley DB with Apache

A mod_db4 Apache module is included in the Berkeley DB distribution, providing a safe framework for running Berkeley DB applications in an Apache 1.3 environment. Apache natively provides no interface for communication between threads or processes, so the mod_db4 module exists to provide this communication.

In general, it is dangerous to run Berkeley DB in a multiprocess system without some facility to coordinate database recovery between processes sharing the database environment after application or system failure. Failure to run recovery after failure can include process hangs and an inability to access the database environment. The mod_db4 Apache module oversees the proper management of Berkeley DB database environment resources. Developers building applications using Berkeley DB as the storage manager within an Apache module should employ this technique for proper resource management.

Specifically, mod_db4 provides the following facilities:

1. New constructors for DB_ENV and DB handles, which install replacement open/close methods.

2. Transparent caching of open DB_ENV and DB handles.

3. Reference counting on all structures, allowing the module to detect the initial opening of any managed database and automatically perform recovery.

4. Automatic detection of unexpected failures (segfaults, or a module actually calling exit() and avoiding shut down phases), and automatic termination of all child processes with open database resources to attempt consistency.

mod_db4 is designed to be used as an alternative interface to Berkeley DB. To have another Apache module (for example, mod_foo) use mod_db4, do not link mod_foo against the Berkeley DB library. In your mod_foo makefile, you should:

```
#include "mod_db4_export.h"
```

and add your Apache include directory to your CPPFLAGS.

In mod_foo, to create a mod_db4 managed DB_ENV handle, use the following:

```
int mod_db4_db_env_create(DB_ENV **dbenvp, u_int32_t flags);
```

which takes identical arguments to db_env_create().

To create a mod_db4 managed DB handle, use the following:

```
int mod_db4_db_create(DB **dbp, DB_ENV *dbenv, u_int32_t flags);
```

which takes identical arguments to db_create().

Otherwise the API is completely consistent with the standard Berkeley DB API.

The mod_db4 module requires the Berkeley DB library be compiled with C++ extensions and the MM library. (The MM library provides an abstraction layer which allows related processes to share data easily. On systems where shared memory or other inter-process communication mechanisms are not available, the MM library emulates them using temporary files. MM is used in several operating systems to provide shared memory pools to Apache modules.)

To build this apache module, perform the following steps:

```
% ./configure --with-apxs=[path to the apxs utility] \
 --with-db4=[Berkeley DB library installation directory] \
 --with-mm=[libmm installation directory]
% make
% make install
```

Post-installation, modules can use this extension via the functions documented in $APACHE_INCLUDEDIR/mod_db4_export.h.

Using Berkeley DB with Perl

The original Perl module for Berkeley DB was DB_File, which was written to interface to Berkeley DB version 1.85. The newer Perl module for Berkeley DB is BerkeleyDB, which was written to interface to version 2.0 and subsequent releases. Because Berkeley DB version 2.X has a compatibility API for version 1.85, you can (and should!) build DB_File using version 2.X of Berkeley DB, although DB_File will still only support the 1.85 functionality.

DB_File is distributed with the standard Perl source distribution (look in the directory "ext/DB_File"). You can find both DB_File and BerkeleyDB on CPAN, the Comprehensive Perl Archive Network of mirrored FTP sites. The master CPAN site is ftp://ftp.funet.fi/.

Versions of both BerkeleyDB and DB_File that are known to work correctly with each release of Berkeley DB are included in the distributed Berkeley DB source tree, in the subdirectories `perl.BerkeleyDB` and `perl.DB_File`. Each of those directories contains a `README` file with instructions on installing and using those modules.

The Perl interface is not maintained by Oracle. Questions about the DB_File and BerkeleyDB modules are best asked on the Usenet newsgroup comp.lang.perl.modules.

Using Berkeley DB with PHP

A PHP 4 extension for this release of Berkeley DB is included in the distribution package. It can either either link directly against the installed Berkeley DB library (which is necessary for running in a non-Apache/mod_php4 environment), or against mod_db4, which provides additional safety when running under Apache/mod_php4.

The PHP extension provides the following classes, which mirror the standard Berkeley DB C++ API.

```
class Db4Env {
 function Db4Env($flags = 0) {}
 function close($flags = 0) {}
```

```
 function dbremove($txn, $filename, $database = null, $flags = 0) {}
 function dbrename($txn, $file, $database, $new_database, $flags = 0) {}
 function open($home, $flags = DB_CREATE  | DB_INIT_LOCK | DB_INIT_LOG | DB_INIT_MPOOL | DB_IN
 function remove($home, $flags = 0) {}
 function set_data_dir($directory) {}
 function txn_begin($parent_txn = null, $flags = 0) {}
 function txn_checkpoint($kbytes, $minutes, $flags = 0) {}
}

class Db4 {
 function Db4($dbenv = null) {} // create a new Db4 object using the optional DbEnv
 function open($txn = null, $file = null, $database = null, $flags = DB_CREATE, $mode = 0) {}
 function close() {}
 function del($key, $txn = null) {}
 function get($key, $txn = null, $flags = 0) {}
 function pget($key, &$pkey, $txn = null, $flags = 0) {}
 function get_type() {} // returns the stringified database type name
 function stat($txn = null, $flags = 0) {} // returns statistics as an as
 function join($cursor_list, $flags = 0) {}
 function sync() {}
 function truncate($txn = null, $flags = 0) {}
 function cursor($txn = null, flags = 0) {}
}

class Db4Txn {
 function abort() {}
 function commit() {}
 function discard() {}
 function id() {}
 function set_timeout($timeout, $flags = 0) {}
}

class Db4Cursor {
 function close() {}
 function count() {}
 function del() {}
 function dup($flags = 0) {}
 function get($key, $flags = 0) {}
 function pget($key, &$primary_key, $flags = 0) {}
 function put($key, $data, $flags = 0) {}
}
```

The PHP extension attempts to be "smart" for you by:

1. Auto-committing operations on transactional databases if no explicit Db4Txn object is specified.

2. Performing reference and dependency checking to insure that all resources are closed in the correct order.

3. Supplying default values for flags.

To install this PHP module linked against the mod_db4 framework, perform the following steps:

```
% phpize
% ./configure --with-db4=[Berkeley DB library installation directory] \
 --with-mod_db4=$APACHE_INCLUDEDIR
% make
% make install
```

Then, in your php.ini file add the following:

```
extension=db4.so
```

This extension will now only run in a SAPI linked into Apache httpd (mod_php4, most likely), and will take advantage of all of its auto-recovery and handle-caching facilities.

To install this php module linked against the Berkeley DB library and not the mod_db4 framework, perform the following steps:

```
% phpize
% ./configure --with-db4=[Berkeley DB library installation directory]
% make
% make install
```

Then in your php.ini file add:

```
extension=db4.so
```

Chapter 22. Dumping and Reloading Databases

The db_dump and db_load utilities

There are three utilities used for dumping and loading Berkeley DB databases: the db_dump utility, the db_dump185 utility and the db_load utility.

The db_dump utility and the db_dump185 utility dump Berkeley DB databases into a flat-text representation of the data that can be read by db_load utility. The only difference between them is that the db_dump utility reads Berkeley DB version 2 and greater database formats, whereas the db_dump185 utility reads Berkeley DB version 1.85 and 1.86 database formats.

The db_load utility reads either the output format used by the dump utilities or (optionally) a flat-text representation created using other tools, and stores it into a Berkeley DB database.

Dumping and reloading Hash databases that use user-defined hash functions will result in new databases that use the default hash function. Although using the default hash function may not be optimal for the new database, it will continue to work correctly.

Dumping and reloading Btree databases that use user-defined prefix or comparison functions will result in new databases that use the default prefix and comparison functions. In this case, it is quite likely that applications will be unable to retrieve records, and it is possible that the load process itself will fail.

The only available workaround for either Hash or Btree databases is to modify the sources for the db_load utility to load the database using the correct hash, prefix, and comparison functions.

Dump output formats

There are two output formats used by the db_dump utility and db_dump185 utility.

In both output formats, the first few lines of the output contain header information describing the underlying access method, filesystem page size, and other bookkeeping information.

The header information starts with a single line, VERSION=N, where N is the version number of the dump output format.

The header information is then output in name=value pairs, where name may be any of the keywords listed in the db_load utility manual page, and value will be its value. Although this header information can be manually edited before the database is reloaded, there is rarely any reason to do so because all of this information can also be specified or overridden by command-line arguments to the db_load utility.

The header information ends with single line HEADER=END.

Following the header information are the key/data pairs from the database. If the database being dumped is a Btree or Hash database, or if the **-k** option was specified, the output will be paired lines of text where the first line of the pair is the key item, and the second line of the pair is its corresponding data item. If the database being dumped is a Queue or Recno

database, and the **-k** option was not specified, the output will be lines of text where each line is the next data item for the database. Each of these lines is preceded by a single space.

If the **-p** option was specified to the db_dump utility or db_dump185 utility, the key/data lines will consist of single characters representing any characters from the database that are *printing characters* and backslash (\) escaped characters for any that were not. Backslash characters appearing in the output mean one of two things: if the backslash character precedes another backslash character, it means that a literal backslash character occurred in the key or data item. If the backslash character precedes any other character, the next two characters must be interpreted as hexadecimal specification of a single character; for example, **\0a** is a newline character in the ASCII character set.

Although some care should be exercised, it is perfectly reasonable to use standard text editors and tools to edit databases dumped using the **-p** option before reloading them using the db_load utility.

Note that the definition of a printing character may vary from system to system, so database representations created using the **-p** option may be less portable than those created without it.

If the **-p** option in not specified to db_dump utility or db_dump185 utility, each output line will consist of paired hexadecimal values; for example, the line **726f6f74** is the string **root** in the ASCII character set.

In all output formats, the key and data items are ended by a single line DATA=END.

Where multiple databases have been dumped from a file, the overall output will repeat; that is, a new set of headers and a new set of data items.

Loading text into databases

The db_load utility can be used to load text into databases. The **-T** option permits nondatabase applications to create flat-text files that are then loaded into databases for fast, highly-concurrent access. For example, the following command loads the standard UNIX `/etc/passwd` file into a database, with the login name as the key item and the entire password entry as the data item:

```
awk -F: '{print $1; print $0}' < /etc/passwd |\
    sed 's/\\/\\\\/g' | db_load -T -t hash passwd.db
```

Note that backslash characters naturally occurring in the text are escaped to avoid interpretation as escape characters by the db_load utility.

Chapter 23. System Installation Notes

File utility /etc/magic information

The file(1) utility is a UNIX utility that examines and classifies files, based on information found in its database of file types, the /etc/magic file. The following information may be added to your system's /etc/magic file to enable file(1) to correctly identify Berkeley DB database files.

The file(1) utility magic(5) information for the standard System V UNIX implementation of the file(1) utility is included in the Berkeley DB distribution for both big-endian [magic.s5.be.txt] (for example, Sparc) and little-endian [magic.s5.le.txt] (for example, x86) architectures.

The file(1) utility magic(5) information for Release 3.X of Ian Darwin's implementation of the file utility (as distributed by FreeBSD and most Linux distributions) is included in the Berkeley DB distribution. This magic.txt information is correct for both big-endian and little-endian architectures.

Building with multiple versions of Berkeley DB

In some cases it may be necessary to build applications which include multiple versions of Berkeley DB. Examples include applications which include software from other vendors, or applications running on a system where the system C library itself uses Berkeley DB. In such cases, the two versions of Berkeley DB may be incompatible, that is, they may have different external and internal interfaces, and may even have different underlying database formats.

To create a Berkeley DB library whose symbols won't collide with other Berkeley DB libraries (or other application or library modules, for that matter), configure Berkeley DB using the --with-uniquename=NAME configuration option, and then build Berkeley DB as usual. (Note that --with-uniquename=NAME only affects the Berkeley DB C language library build; loading multiple versions of the C++ or Java APIs will require additional work.) The modified symbol names are hidden from the application in the Berkeley DB header files, that is, there is no need for the application to be aware that it is using a special library build as long as it includes the appropriate Berkeley DB header file.

If "NAME" is not specified when configuring with --with-uniquename=NAME, a default value built from the major and minor numbers of the Berkeley DB release will be used. It is rarely necessary to specify NAME; using the major and minor release numbers will ensure that only one copy of the library will be loaded into the application unless two distinct versions really are necessary.

When distributing any library software that uses Berkeley DB, or any software which will be recompiled by users for their systems, we recommend two things: First, include the Berkeley DB release as part of your release. This will insulate your software from potential Berkeley DB API changes as well as simplifying your coding because you will only have to code to a single version of the Berkeley DB API instead of adapting at compile time to whatever version of Berkeley DB happens to be installed on the target system. Second, use --with-uniquename=NAME

when configuring Berkeley DB, because that will insure that you do not unexpectedly collide with other application code or a library already installed on the target system.

Chapter 24. Debugging Applications

Introduction to debugging

Because Berkeley DB is an embedded library, debugging applications that use Berkeley DB is both harder and easier than debugging a separate server. Debugging can be harder because when a problem arises, it is not always readily apparent whether the problem is in the application, is in the database library, or is a result of an unexpected interaction between the two. Debugging can be easier because it is easier to track down a problem when you can review a stack trace rather than deciphering interprocess communication messages. This chapter is intended to assist you with debugging applications and reporting bugs to us so that we can provide you with the correct answer or fix as quickly as possible.

When you encounter a problem, there are a few general actions you can take:

Review the Berkeley DB error output:

If an error output mechanism has been configured in the Berkeley DB environment, additional run-time error messages are made available to the applications. If you are not using an environment, it is well worth modifying your application to create one so that you can get more detailed error messages. See Run-time error information (page 279) for more information on configuring Berkeley DB to output these error messages.

Review DB_ENV->set_verbose() function

, and see if any of them will produce additional information that might help understand the problem.

Add run-time diagnostics:

You can configure and build Berkeley DB to perform run-time diagnostics. (By default, these checks are not done because they can seriously impact performance.) See Compile-time configuration (page 278) for more information.

Apply all available patches:

Before reporting a problem in Berkeley DB, please upgrade to the latest Berkeley DB release, if possible, or at least make sure you have applied any updates available for your release from the Berkeley DB web site [http://www.oracle.com/technology/software/products/berkeley-db/db/index.html].

Run the test suite:

If you see repeated failures or failures of simple test cases, run the Berkeley DB test suite to determine whether the distribution of Berkeley DB you are using was built and configured correctly.

Compile-time configuration

There are three compile-time configuration options that assist in debugging Berkeley DB and Berkeley DB applications:

--enable-debug

If you want to build Berkeley DB with **-g** as the C and C++ compiler flag, enter --enable-debug as an argument to configure. This will create Berkeley DB with debugging

symbols, as well as load various Berkeley DB routines that can be called directly from a debugger to display database page content, cursor queues, and so forth. (Note that the **-O** optimization flag will still be specified. To compile with only the **-g**, explicitly set the `CFLAGS` environment variable before configuring.)

--enable-diagnostic

If you want to build Berkeley DB with debugging run-time sanity checks and with DIAGNOSTIC #defined during compilation, enter --enable-diagnostic as an argument to configure. This will cause a number of special checks to be performed when Berkeley DB is running. This flag should not be defined when configuring to build production binaries because it degrades performance.

--enable-umrw

When compiling Berkeley DB for use in run-time memory consistency checkers (in particular, programs that look for reads and writes of uninitialized memory), use --enable-umrw as an argument to configure. This guarantees, among other things, that Berkeley DB will completely initialize allocated pages rather than initializing only the minimum necessary amount.

Run-time error information

Normally, when an error occurs in the Berkeley DB library, an integer value (either a Berkeley DB specific value or a system `errno` value) is returned by Berkeley DB. In some cases, however, this value may be insufficient to completely describe the cause of the error, especially during initial application debugging.

Most Berkeley DB errors will result in additional information being written to a standard file descriptor or output stream. Additionally, Berkeley DB can be configured to pass these verbose error messages to an application function. There are four methods intended to provide applications with additional error information: DB_ENV->set_errcall(), DB_ENV->set_errfile(), DB_ENV->set_errpfx() and DB_ENV->set_verbose().

The Berkeley DB error-reporting facilities do not slow performance or significantly increase application size, and may be run during normal operation as well as during debugging. Where possible, we recommend these options always be configured and the output saved in the filesystem. We have found that this often saves time when debugging installation or other system-integration problems.

In addition, there are three methods to assist applications in displaying their own error messages: db_strerror(), DB_ENV->err(), and `DB_ENV->errx()`. The first is a superset of the ANSI C strerror function, and returns a descriptive string for any error return from the Berkeley DB library. The DB_ENV->err() and `DB_ENV->errx()` methods use the error message configuration options described previously to format and display error messages to appropriate output devices.

Reviewing Berkeley DB log files

If you are running with transactions and logging, the db_printlog utility can be a useful debugging aid. The db_printlog utility will display the contents of your log files in a human readable (and machine-readable) format.

The db_printlog utility will attempt to display any and all log files present in a designated db_home directory. For each log record, the db_printlog utility will display a line of the form:

```
[22][28]db_big: rec: 43 txnid 80000963 prevlsn [21][10483281]
```

The opening numbers in square brackets are the *log sequence number* (*LSN*) of the log record being displayed. The first number indicates the log file in which the record appears, and the second number indicates the offset in that file of the record.

The first character string identifies the particular log operation being reported. The log records corresponding to particular operations are described following. The rest of the line consists of name/value pairs.

The rec field indicates the record type (this is used to dispatch records in the log to appropriate recovery functions).

The txnid field identifies the transaction for which this record was written. A txnid of 0 means that the record was written outside the context of any transaction. You will see these most frequently for checkpoints.

Finally, the prevlsn contains the LSN of the last record for this transaction. By following prevlsn fields, you can accumulate all the updates for a particular transaction. During normal abort processing, this field is used to quickly access all the records for a particular transaction.

After the initial line identifying the record type, each field of the log record is displayed, one item per line. There are several fields that appear in many different records and a few fields that appear only in some records.

The following table presents each currently written log record type with a brief description of the operation it describes. Any of these record types may have the string "_debug" appended if they were written because DB_TXN_NOT_DURABLE was specified and the system was configured with --enable-diagnostic.

Log Record Type	Description
bam_adj	Used when we insert/remove an index into/from the page header of a Btree page.
bam_cadjust	Keeps track of record counts in a Btree or Recno database.
bam_cdel	Used to mark a record on a page as deleted.
bam_curadj	Used to adjust a cursor location when a nearby record changes in a Btree database.
bam_merge	Used to merge two Btree database pages during compaction.
bam_pgno	Used to replace a page number in a Btree record.
bam_rcuradj	Used to adjust a cursor location when a nearby record changes in a Recno database.

Log Record Type	Description
bam_relink	Fix leaf page prev/next chain when a page is removed.
bam_repl	Describes a replace operation on a record.
bam_root	Describes an assignment of a root page.
bam_rsplit	Describes a reverse page split.
bam_split	Describes a page split.
crdel_inmem_create	Record the creation of an in-memory named database.
crdel_inmem_remove	Record the removal of an in-memory named database.
crdel_inmem_rename	Record the rename of an in-memory named database.
crdel_metasub	Describes the creation of a metadata page for a subdatabase.
db_addrem	Add or remove an item from a page of duplicates.
db_big	Add an item to an overflow page (*overflow pages* contain items too large to place on the main page)
db_cksum	Unable to checksum a page.
db_debug	Log debugging message.
db_noop	This marks an operation that did nothing but update the LSN on a page.
db_ovref	Increment or decrement the reference count for a big item.
db_pg_alloc	Indicates we allocated a page to a database.
db_pg_free	Indicates we freed a page (freed pages are added to a freelist and reused).
db_pg_freedata	Indicates we freed a page that still contained data entries (freed pages are added to a freelist and reused.)
db_pg_init	Indicates we reinitialized a page during a truncate.
db_pg_sort	Sort the free page list and free pages at the end of the file.
dbreg_register	Records an open of a file (mapping the filename to a log-id that is used in subsequent log operations).
fop_create	Create a file in the file system.

Log Record Type	Description
fop_file_remove	Remove a name in the file system.
fop_remove	Remove a file in the file system.
fop_rename	Rename a file in the file system.
fop_write	Write bytes to an object in the file system.
ham_chgpg	Used to adjust a cursor location when a Hash page is removed, and its elements are moved to a different Hash page.
ham_copypage	Used when we empty a bucket page, but there are overflow pages for the bucket; one needs to be copied back into the actual bucket.
ham_curadj	Used to adjust a cursor location when a nearby record changes in a Hash database.
ham_groupalloc	Allocate some number of contiguous pages to the Hash database.
ham_insdel	Insert/delete an item on a Hash page.
ham_metagroup	Update the metadata page to reflect the allocation of a sequence of contiguous pages.
ham_newpage	Adds or removes overflow pages from a Hash bucket.
ham_replace	Handle updates to records that are on the main page.
ham_splitdata	Record the page data for a split.
qam_add	Describes the actual addition of a new record to a Queue.
qam_del	Delete a record in a Queue.
qam_delext	Delete a record in a Queue with extents.
qam_incfirst	Increments the record number that refers to the first record in the database.
qam_mvptr	Indicates we changed the reference to either or both of the first and current records in the file.
txn_child	Commit a child transaction.
txn_ckp	Transaction checkpoint.
txn_recycle	Transaction IDs wrapped.
txn_regop	Logs a regular (non-child) transaction commit.
txn_xa_regop	Logs a prepare message.

Augmenting the Log for Debugging

When debugging applications, it is sometimes useful to log not only the actual operations that modify pages, but also the underlying Berkeley DB functions being executed. This form of logging can add significant bulk to your log, but can permit debugging application errors that are almost impossible to find any other way. To turn on these log messages, specify the --enable-debug_rop and --enable-debug_wop configuration options when configuring Berkeley DB. See Configuring Berkeley DB (page 290) for more information.

Extracting Committed Transactions and Transaction Status

Sometimes, it is helpful to use the human-readable log output to determine which transactions committed and aborted. The awk script, commit.awk, (found in the db_printlog directory of the Berkeley DB distribution) allows you to do just that. The following command, where log_output is the output of db_printlog, will display a list of the transaction IDs of all committed transactions found in the log:

```
awk -f commit.awk log_output
```

If you need a complete list of both committed and aborted transactions, then the script status.awk will produce it. The syntax is as follows:

```
awk -f status.awk log_output
```

Extracting Transaction Histories

Another useful debugging aid is to print out the complete history of a transaction. The awk script txn.awk allows you to do that. The following command line, where log_output is the output of the db_printlog utility and txnlist is a comma-separated list of transaction IDs, will display all log records associated with the designated transaction ids:

```
awk -f txn.awk TXN=txnlist log_output
```

Extracting File Histories

The awk script fileid.awk allows you to extract all log records that refer to a designated file. The syntax for the fileid.awk script is the following, where log_output is the output of db_printlog and fids is a comma-separated list of fileids:

```
awk -f fileid.awk PGNO=fids log_output
```

Extracting Page Histories

The awk script pgno.awk allows you to extract all log records that refer to designated page numbers. However, because this script will extract records with the designated page numbers for all files, it is most useful in conjunction with the fileid script. The syntax for the pgno.awk script is the following, where log_output is the output of db_printlog and pgnolist is a comma-separated list of page numbers:

```
awk -f pgno.awk PGNO=pgnolist log_output
```

Other log processing tools

The awk script count.awk prints out the number of log records encountered that belonged to some transaction (that is, the number of log records excluding those for checkpoints and non-transaction-protected operations).

The script range.awk will extract a subset of a log. This is useful when the output of db_printlog utility is too large to be reasonably manipulated with an editor or other tool. The syntax for range.awk is the following, where **sf** and **so** represent the LSN of the beginning of the sublog you want to extract, and **ef** and **eo** represent the LSN of the end of the sublog you want to extract:

```
awk -f range.awk START_FILE=sf START_OFFSET=so END_FILE=ef END_OFFSET=eo log_output
```

Chapter 25. Building Berkeley DB for the BREW simulator

This chapter has general instructions for building the Berkeley DB library and applets for the BREW platform.

The `build_brew` directory in the Berkeley DB distribution contains project files for Microsoft Visual C++:

Project File	Description
bdb_brew.dsw	Visual C++ 6.0 workspace
bdb_brew.dsp	Visual C++ 6.0 project

The project file can be used to build the Berkeley DB library for the BREW platform. Both BREW SDK versions 2 and 3 are supported. By default, the build is for BREW SDK version 2.

The steps for building the Berkeley DB library for the BREW Simulator are as follows:

1. Install the BREW SDK and BREW SDK Tools.

2. To build for BREW SDK version 3, edit the Berkeley DB source distribution file `build_brew/db_config.h` file and **remove** the line "#define HAVE_BREW_SDK2". No changes are required to build for BREW SDK version 2.

3. Select *File -> Open Workspace*. Look in the build_brew directory for Workspaces, select *bdb_brew.dsw*, and select *Open*.

4. The BREW SDK creates an environment entry "BREWDIR" after the SDK installation. Confirm the entry exists, and if not, create an environment entry which points to the BREW SDK's root directory.

5. For the BREW platform, only the project "bdb_brew" is available. Set the bdb_brew project as the active project and select the appropriate option for the build_all project (Debug or Release). Then select *OK*.

6. To build, press F7.

The build results are placed in a subdirectory of build_brew named after the selected configuration (for example, `build_brew\Release` or `build_brew\Debug`).

When building the application during development, you should normally use compile options "Debug Multithreaded DLL" and link against `build_brew\Debug\bdb_brew.lib`. You can also build using a release version of the Berkeley DB libraries and tools, which will be placed in `build_brew\Release\bdb_brew.lib`. When linking against the release build, you should compile your code with the "Release Multithreaded DLL" compile option. You will also need to add the `build_brew` directory to the list of include directories of your application's project, or copy the Berkeley DB include files to another location.

Building a BREW applet with Berkeley DB library

Building a Berkeley DB application in the BREW environment is similar to building in a Windows environment. Ensure that `db.h` is in the build include path and `bdb_brew.lib` is in the build library path (alternatively, you can add project dependencies to the `bdb_brew` project).

BREW applets require a few minor additions:

1. The **BDBApp** structure must be extended -- the extended definition of BDBApp is found in the include file `db.h`.

2. Before any Berkeley DB operations are performed, the applet must call the "int brew_bdb_begin(void)" function. This function returns 0 on success, and non-zero on failure.

3. After the last Berkeley DB operation is performed, the applet must call the "void brew_bdb_end(void)" function. Note that the brew_bdb_end function cannot be called in the applet "cleanup" function. If that is a requirement, use the following code instead:

```
BDBApp *pMe=(BDBApp *)po;
if (pMe->db_global_values != NULL)
 FREE(pMe->db_global_values);
```

Building a BREW applet for the physical device

The binaries linked with the Berkeley DB library for the BREW simulator are not target files that can run on the physical device. In order to build for the physical device, an ARM compiler is needed: the recommended ARM compiler is ARM Developer Suite 1.2.

The steps for building a BREW applet for the physical device are as follows:

1. Set the target BREW Applet project as the active project.

2. Select "Generate ARM Make file" in the BREW tool bar for VC6, and a make file will be generated (if this step does not work, confirm your ADS was correctly installed).

3. The Berkeley DB library must then be manually added to this make file. See the `build_brew\bdbread.mak` file in the Berkeley DB distribution for an example.

4. Select *Tools -> BREW Application 'Make'* to build.

The target .mod file will be created in the build directory, and this is the file which should be uploaded to the physical device.

Chapter 26. Building Berkeley DB for S60

This page has general instructions for building the Berkeley DB library and applications for the S60 platform.

Building Berkeley DB for S60 requires S60 SDK version 3 or above with Symbian V9.1 or above; the 3rd Edition FP1 SDK is recommended.

The NOKIA OpenC plugin should be installed on both emulator and device.

CodeWarrior IDE for S60 V3.1 is recommended.

Building Berkeley DB for the S60 Emulator

The `build_s60` directory in the Berkeley DB distribution contains a `bdb_s60.mmp` project file, which can be imported to CodeWarrior or Carbide, etc. This project file can be used to build the Berkeley DB library for the S60 platform.

The steps for building the Berkeley DB library for the S60 emulator are as follows:

1. Install the S60 SDK and OpenC library.
2. In CodeWarrior, select *File -> Import project from .mmp file*.
3. Select an SDK to use with this project: *nokia ->S60 S60_3rd_FP1*.
4. Click to browse for MMP file selection. Look in the `build_s60` directory for Workspaces and select *bdb_s60.mmp*.
5. Click Next.
6. Click Finish.
7. Select build target to WINSCW UDEB, to build, press F7.

The build will create the file `bdb_s60.lib`, which is installed into `$EPOCROOT/epoc32/release/winscw/udeb`, which can be linked by Berkeley DB applications for the emulator.

Building Berkeley DB Library for the Device

1. Install the CSL Arm tool chain.
2. From `$EPOCROOT/epoc32/release/armv5/lib`, add OpenC library libpthread.dso to the GCCE library(libc.dso already added).
3. Since Berkeley DB uses old-style function definitions, it will not be accepted by the default compiler:arm-none-symbianelf-g++.exe. Change the compiler to arm-none-symbianelf-gcc.exe and specify the "-x c" option there.
4. Select build target to GCCE UREL, to build, press F7.

The build will create the file `bdb_s60.lib`, which is installed into `$EPOCROOT/epoc32/release/armv5/urel`, which can be linked by Berkeley DB applications for device.

Building a S60 application with the Berkeley DB library

Building a Berkeley DB application in the S60 environment is similar to building in a Windows environment. Ensure the include file `build_s60/db.h` is in the build include path and the created `bdb_s60.lib` is in the build library path.

S60 notes

1. The stack size on the S60 is small by default, and the application will silently fail if the stack is too small. Setting EPOCSTACKSIZE to an appropriate value in the application's mmp file will resolve this problem.

2. The Berkeley DB build on the S60 is a "small build", disabling some of the Berkeley DB library features. This build is equivalent to the **--enable-smallbuild** configuration option described in Building a small memory footprint library (page 294).

Chapter 27. Building Berkeley DB for UNIX/POSIX

Building for UNIX/POSIX

The Berkeley DB distribution builds up to four separate libraries: the base C API Berkeley DB library and the optional C++, Java, and Tcl API libraries. For portability reasons, each library is standalone and contains the full Berkeley DB support necessary to build applications; that is, the C++ API Berkeley DB library does not require any other Berkeley DB libraries to build and run C++ applications.

Building for Linux, Mac OS X and the QNX Neutrino release is the same as building for a conventional UNIX platform.

The Berkeley DB distribution uses the Free Software Foundation's autoconf [http://www.gnu.org/software/autoconf/autoconf.html] and libtool [http://www.gnu.org/software/libtool/libtool.html] tools to build on UNIX platforms. In general, the standard configuration and installation options for these tools apply to the Berkeley DB distribution.

To do a standard UNIX build of Berkeley DB, change to the **build_unix** directory and then enter the following two commands:

```
../dist/configure
make
```

This will build the Berkeley DB library.

To install the Berkeley DB library, enter the following command:

```
make install
```

To rebuild Berkeley DB, enter:

```
make clean
make
```

If you change your mind about how Berkeley DB is to be configured, you must start from scratch by entering the following command:

```
make realclean
../dist/configure
make
```

To uninstall Berkeley DB, enter:

```
make uninstall
```

To build multiple UNIX versions of Berkeley DB in the same source tree, create a new directory at the same level as the build_unix directory, and then configure and build in that directory as described previously.

Configuring Berkeley DB

There are several arguments you can specify when configuring Berkeley DB. Although only the Berkeley DB-specific ones are described here, most of the standard GNU autoconf arguments are available and supported. To see a complete list of possible arguments, specify the --help flag to the configure program.

The Berkeley DB specific arguments are as follows:

- **--disable-largefile**

  Some systems, notably versions of HP/UX and Solaris, require special compile-time options in order to create files larger than 2^32 bytes. These options are automatically enabled when Berkeley DB is compiled. For this reason, binaries built on current versions of these systems may not run on earlier versions of the system because the library and system calls necessary for large files are not available. To disable building with these compile-time options, enter --disable-largefile as an argument to configure.

- **--disable-shared, --disable-static**

  On systems supporting shared libraries, Berkeley DB builds both static and shared libraries by default. (Shared libraries are built using the GNU Project's Libtool [http://www.gnu.org/software/libtool/libtool.html] distribution, which supports shared library builds on many (although not all) systems.) To not build shared libraries, configure using the --disable-shared argument. To not build static libraries, configure using the --disable-static argument.

- **--enable-compat185**

  To compile or load Berkeley DB 1.85 applications against this release of the Berkeley DB library, enter --enable-compat185 as an argument to configure. This will include Berkeley DB 1.85 API compatibility code in the library.

- **--enable-cxx**

  To build the Berkeley DB C++ API, enter --enable-cxx as an argument to configure.

- **--enable-debug**

  To build Berkeley DB with **-g** as a compiler flag and with **DEBUG** #defined during compilation, enter --enable-debug as an argument to configure. This will create a Berkeley DB library and utilities with debugging symbols, as well as load various routines that can be called from a debugger to display pages, cursor queues, and so forth. If installed, the utilities will not be stripped. This argument should not be specified when configuring to build production binaries.

- **--enable-debug_rop**

  To build Berkeley DB to output log records for read operations, enter --enable-debug_rop as an argument to configure. This argument should not be specified when configuring to build production binaries.

- **--enable-debug_wop**

To build Berkeley DB to output log records for write operations, enter --enable-debug_wop as an argument to configure. This argument should not be specified when configuring to build production binaries.

- **--enable-diagnostic**

  To build Berkeley DB with run-time debugging checks, enter --enable-diagnostic as an argument to configure. This causes a number of additional checks to be performed when Berkeley DB is running, and also causes some failures to trigger process abort rather than returning errors to the application. Applications built using this argument should not share database environments with applications built without this argument. This argument should not be specified when configuring to build production binaries.

- **--enable-dump185**

  To convert Berkeley DB 1.85 (or earlier) databases to this release of Berkeley DB, enter --enable-dump185 as an argument to configure. This will build the db_dump185 utility, which can dump Berkeley DB 1.85 and 1.86 databases in a format readable by the Berkeley DB db_load utility.

  The system libraries with which you are loading the db_dump185 utility must already contain the Berkeley DB 1.85 library routines for this to work because the Berkeley DB distribution does not include them. If you are using a non-standard library for the Berkeley DB 1.85 library routines, you will have to change the Makefile that the configuration step creates to load the db_dump185 utility with that library.

- **--enable-java**

  To build the Berkeley DB Java API, enter --enable-java as an argument to configure. To build Java, you must also build with shared libraries. Before configuring, you must set your PATH environment variable to include javac. Note that it is not sufficient to include a symbolic link to javac in your PATH because the configuration process uses the location of javac to determine the location of the Java include files (for example, jni.h). On some systems, additional include directories may be needed to process jni.h; see Changing compile or load options (page 295) for more information.

- **--enable-posixmutexes**

  To force Berkeley DB to use the POSIX pthread mutex interfaces for underlying mutex support, enter --enable-posixmutexes as an argument to configure. This is rarely necessary: POSIX mutexes will be selected automatically on systems where they are the preferred implementation.

  The --enable-posixmutexes configuration argument is normally used in two ways: First, when there are multiple mutex implementations available and the POSIX mutex implementation is not the preferred one (for example, on Solaris where the LWP mutexes are used by default). Second, by default the Berkeley DB library will only select the POSIX mutex implementation if it supports mutexes shared between multiple processes, as described for the pthread_condattr_setpshared and pthread_mutexattr_setpshared interfaces. The --enable-posixmutexes configuration argument can be used to force the selection of POSIX

mutexes in this case, which can improve application performance significantly when the alternative mutex implementation is a non-blocking one (for example test-and-set assembly instructions). However, configuring to use POSIX mutexes when the implementation does not have inter-process support will only allow the creation of private database environments, that is, environments where the DB_PRIVATE flag is specified to the DB_ENV->open() method.

Specifying the --enable-posixmutexes configuration argument may require that applications and Berkeley DB be linked with the -lpthread library.

- **--enable-pthread_api**

  To configure Berkeley DB for a POSIX pthreads application (with the exception that POSIX pthread mutexes may not be selected as the underlying mutex implementation for the build), enter --enable-pthread_api as an argument to configure. The build will include the Berkeley DB replication manager interfaces and will use the POSIX standard pthread_self and pthread_yield functions to identify threads of control and yield the processor. The --enable-pthread_api argument requires POSIX pthread support already be installed on your system.

  Specifying the --enable-pthread_api configuration argument may require that applications and Berkeley DB be linked with the -lpthread library.

- **--enable-smallbuild**

  To build a small memory footprint version of the Berkeley DB library, enter --enable-smallbuild as an argument to configure. The --enable-smallbuild argument is equivalent to individually specifying --disable-cryptography, --disable-hash, --disable-queue, --disable-replication, --disable-statistics and --disable-verify, turning off cryptography support, the Hash and Queue access methods, database environment replication support and database verification support. See Building a small memory footprint library (page 294) for more information.

- **--enable-stl**

  To build the Berkeley DB C++ STL API, enter --enable-stl as an argument to configure. Setting this argument implies that --enable-cxx is set, and the Berkeley DB C++ API will be built too.

  There will be a libdb_stl-X.X.a and libdb_stl-X.X.so built, which are the static and shared library you should link your application with in order to make use of Berkeley DB via its STL API.

  If your compiler is not ISO C++ compliant, the configure may fail with this argument specified because the STL API requires standard C++ template features. In this case, you will need a standard C++ compiler. So far gcc is the best choice, we have tested and found that gcc-3.4.4 and all its newer versions can build the Berkeley DB C++ STL API successfully.

  And you need to include the STL API header files in your application code. If you are using the Berkeley DB source tree, the header files are in <Berkeley DB Source Root >/stl directory; If you are using the installed version, these header files are in < Berkeley DB Installed Directory>/include, as well as the db.h and db_cxx.h header files.

- **--enable-tcl**

To build the Berkeley DB Tcl API, enter --enable-tcl as an argument to configure. This configuration argument expects to find Tcl's tclConfig.sh file in the `/usr/local/lib` directory. See the --with-tcl argument for instructions on specifying a non-standard location for the Tcl installation. See Loading Berkeley DB with Tcl (page 266) for information on sites from which you can download Tcl and which Tcl versions are compatible with Berkeley DB. To build Tcl, you must also build with shared libraries.

- **--enable-test**

  To build the Berkeley DB test suite, enter --enable-test as an argument to configure. To run the Berkeley DB test suite, you must also build the Tcl API. This argument should not be specified when configuring to build production binaries.

- **--enable-uimutexes**

  To force Berkeley DB to use the UNIX International (UI) mutex interfaces for underlying mutex support, enter --enable-uimutexes as an argument to configure. This is rarely necessary: UI mutexes will be selected automatically on systems where they are the preferred implementation.

  The --enable-uimutexes configuration argument is normally used when there are multiple mutex implementations available and the UI mutex implementation is not the preferred one (for example, on Solaris where the LWP mutexes are used by default).

  Specifying the --enable-uimutexes configuration argument may require that applications and Berkeley DB be linked with the -lthread library.

- **--enable-umrw**

  Rational Software's Purify product and other run-time tools complain about uninitialized reads/writes of structure fields whose only purpose is padding, as well as when heap memory that was never initialized is written to disk. Specify the --enable-umrw argument during configuration to mask these errors. This argument should not be specified when configuring to build production binaries.

- **--with-mutex=MUTEX**

  To force Berkeley DB to use a specific mutex implementation, configure with --with-mutex=MUTEX, where MUTEX is the mutex implementation you want. For example, --with-mutex=x86/gcc-assembly will configure Berkeley DB to use the x86 GNU gcc compiler based test-and-set assembly mutexes. This is rarely necessary and should be done only when the default configuration selects the wrong mutex implementation. A list of available mutex implementations can be found in the distribution file `dist/aclocal/mutex.m4`.

- **--with-tcl=DIR**

  To build the Berkeley DB Tcl API, enter --with-tcl=DIR, replacing DIR with the directory in which the Tcl tclConfig.sh file may be found. See Loading Berkeley DB with Tcl (page 266) for information on sites from which you can download Tcl and which Tcl versions are compatible with Berkeley DB. To build Tcl, you must also build with shared libraries.

- **--with-uniquename=NAME**

  To build Berkeley DB with unique symbol names (in order to avoid conflicts with other application modules or libraries), enter --with-uniquename=NAME, replacing NAME with a string that to be appended to every Berkeley DB symbol. If "=NAME" is not specified, a default value of "_MAJORMINOR" is used, where MAJORMINOR is the major and minor release numbers of the Berkeley DB release. See Building with multiple versions of Berkeley DB (page 276) for more information.

Building a small memory footprint library

There are a set of configuration options to assist you in building a small memory footprint library. These configuration options turn off specific functionality in the Berkeley DB library, reducing the code size. These configuration options include:

To build Berkeley DB without support for cryptography, enter --disable-cryptography as an argument to configure.

To build Berkeley DB without support for the Hash access method, enter --disable-hash as an argument to configure.

To build Berkeley DB without support for the Queue access method, enter --disable-queue as an argument to configure.

To build Berkeley DB without support for the database environment replication, enter --disable-replication as an argument to configure.

To build Berkeley DB without support for the statistics interfaces, enter --disable-statistics as an argument to configure.

To build Berkeley DB without support for database verification, enter --disable-verify as an argument to configure.

Equivalent to individually specifying --disable-cryptography, --disable-hash, --disable-queue, --disable-replication, --disable-statistics and --disable-verify. In addition, when compiling building with the GNU gcc compiler, the --enable-smallbuild option uses the -Os compiler build flag instead of the default -O3.

The following configuration options will increase the size of the Berkeley DB library dramatically and are only useful when debugging applications:

--enable-debug

Build Berkeley DB with symbols for debugging.

--enable-debug_rop

Build Berkeley DB with read-operation logging.

--enable-debug_wop

Build Berkeley DB with write-operation logging.

--enable-diagnostic

Build Berkeley DB with run-time debugging checks.

In addition, static libraries are usually smaller than shared libraries. By default Berkeley DB will build both shared and static libraries. To build only a static library, configure Berkeley DB with the Configuring Berkeley DB (page 290) option.

The size of the Berkeley DB library varies depending on the compiler, machine architecture, and configuration options. As an estimate, production Berkeley DB libraries built with GNU gcc version 3.X compilers have footprints in the range of 400KB to 1.2MB on 32-bit x86 architectures, and in the range of 500KB to 1.4MB on 64-bit x86 architectures.

For assistance in further reducing the size of the Berkeley DB library, or in building small memory footprint libraries on other systems, please contact Berkeley DB support.

Changing compile or load options

You can specify compiler and/or compile and load time flags by using environment variables during Berkeley DB configuration. For example, if you want to use a specific compiler, specify the CC environment variable before running configure:

```
prompt: env CC=gcc ../dist/configure
```

Using anything other than the native compiler will almost certainly mean that you'll want to check the flags specified to the compiler and loader, too.

To specify debugging and optimization options for the C compiler, use the CFLAGS environment variable:

```
prompt: env CFLAGS=-O2 ../dist/configure
```

To specify header file search directories and other miscellaneous options for the C preprocessor and compiler, use the CPPFLAGS environment variable:

```
prompt: env CPPFLAGS=-I/usr/contrib/include ../dist/configure
```

To specify debugging and optimization options for the C++ compiler, use the CXXFLAGS environment variable:

```
prompt: env CXXFLAGS=-Woverloaded-virtual ../dist/configure
```

To specify miscellaneous options or additional library directories for the linker, use the LDFLAGS environment variable:

```
prompt: env LDFLAGS="-N32 -L/usr/local/lib" ../dist/configure
```

If you want to specify additional libraries, set the LIBS environment variable before running configure. For example, the following would specify two additional libraries to load, "posix" and "socket":

```
prompt: env LIBS="-lposix -lsocket" ../dist/configure
```

Make sure that you prepend -L to any library directory names and that you prepend -I to any include file directory names! Also, if the arguments you specify contain blank or tab characters, be sure to quote them as shown previously; that is with single or double quotes around the values you are specifying for LIBS.

The env command, which is available on most systems, simply sets one or more environment variables before running a command. If the env command is not available to you, you can set the environment variables in your shell before running configure. For example, in sh or ksh, you could do the following:

```
prompt: LIBS="-lposix -lsocket" ../dist/configure
```

In csh or tcsh, you could do the following:

```
prompt: setenv LIBS "-lposix -lsocket"
prompt: ../dist/configure
```

See your command shell's manual page for further information.

Installing Berkeley DB

Berkeley DB installs the following files into the following locations, with the following default values:

Configuration Variables	Default value
--prefix	/usr/local/BerkeleyDB.**Major**.**Minor**
--exec_prefix	$(prefix)
--bindir	$(exec_prefix)/bin
--includedir	$(prefix)/include
--libdir	$(exec_prefix)/lib
docdir	$(prefix)/docs

Files	Default location
include files	$(includedir)
libraries	$(libdir)
utilities	$(bindir)
documentation	$(docdir)

With one exception, this follows the GNU Autoconf and GNU Coding Standards installation guidelines; please see that documentation for more information and rationale.

The single exception is the Berkeley DB documentation. The Berkeley DB documentation is provided in HTML format, not in UNIX-style man or GNU info format. For this reason, Berkeley DB configuration does not support **--infodir** or **--mandir**. To change the default installation location for the Berkeley DB documentation, modify the Makefile variable, **docdir**.

When installing Berkeley DB on filesystems shared by machines of different architectures, please note that although Berkeley DB include files are installed based on the value of $(prefix), rather than $(exec_prefix), the Berkeley DB include files are not always architecture independent.

To move the entire installation tree to somewhere besides **/usr/local**, change the value of **prefix**.

To move the binaries and libraries to a different location, change the value of **exec_prefix**. The values of **includedir** and **libdir** may be similarly changed.

Any of these values except for **docdir** may be set as part of the configuration:

```
prompt: ../dist/configure --bindir=/usr/local/bin
```

Any of these values, including **docdir**, may be changed when doing the install itself:

```
prompt: make prefix=/usr/contrib/bdb install
```

The Berkeley DB installation process will attempt to create any directories that do not already exist on the system.

Dynamic shared libraries

Warning: the following information is intended to be generic and is likely to be correct for most UNIX systems. Unfortunately, dynamic shared libraries are not standard between UNIX systems, so there may be information here that is not correct for your system. If you have problems, consult your compiler and linker manual pages, or your system administrator.

The Berkeley DB dynamic shared libraries are created with the name libdb-**major**.**minor**.so, where **major** is the major version number and **minor** is the minor version number. Other shared libraries are created if Java and Tcl support are enabled: specifically, libdb_java-**major**.**minor**.so and libdb_tcl-**major**.**minor**.so.

On most UNIX systems, when any shared library is created, the linker stamps it with a "SONAME". In the case of Berkeley DB, the SONAME is libdb-**major**.**minor**.so. It is important to realize that applications linked against a shared library remember the SONAMEs of the libraries they use and not the underlying names in the filesystem.

When the Berkeley DB shared library is installed, links are created in the install lib directory so that libdb-**major**.**minor**.so, libdb-**major**.so, and libdb.so all refer to the same library. This library will have an SONAME of libdb-**major**.**minor**.so.

Any previous versions of the Berkeley DB libraries that are present in the install directory (such as libdb-2.7.so or libdb-2.so) are left unchanged. (Removing or moving old shared libraries is one drastic way to identify applications that have been linked against those vintage releases.)

Once you have installed the Berkeley DB libraries, unless they are installed in a directory where the linker normally looks for shared libraries, you will need to specify the installation directory as part of compiling and linking against Berkeley DB. Consult your system manuals or system administrator for ways to specify a shared library directory when compiling and linking applications with the Berkeley DB libraries. Many systems support environment variables (for example, LD_LIBRARY_PATH or LD_RUN_PATH), or system configuration files (for example, /etc/ld.so.conf) for this purpose.

Warning: some UNIX installations may have an already existing /usr/lib/libdb.so, and this library may be an incompatible version of Berkeley DB.

We recommend that applications link against libdb.so (for example, using -ldb). Even though the linker uses the file named libdb.so, the executable file for the application remembers the library's SONAME (libdb-**major**.**minor**.so). This has the effect of marking the applications with the versions they need at link time. Because applications locate their needed SONAMEs when they are executed, all previously linked applications will continue to run using the library they were linked with, even when a new version of Berkeley DB is installed and the file libdb.so is replaced with a new version.

Applications that know they are using features specific to a particular Berkeley DB release can be linked to that release. For example, an application wanting to link to Berkeley DB major release "3" can link using -ldb-3, and applications that know about a particular minor release number can specify both major and minor release numbers; for example, -ldb-3.5.

If you want to link with Berkeley DB before performing library installation, the "make" command will have created a shared library object in the .libs subdirectory of the build directory, such as build_unix/.libs/libdb-major.minor.so. If you want to link a file against this library, with, for example, a major number of "3" and a minor number of "5", you should be able to do something like the following:

```
cc -L BUILD_DIRECTORY/.libs -o testprog testprog.o -ldb-3.5
env LD_LIBRARY_PATH="BUILD_DIRECTORY/.libs:$LD_LIBRARY_PATH" ./testprog
```

where **BUILD_DIRECTORY** is the full directory path to the directory where you built Berkeley DB.

The libtool program (which is configured in the build directory) can be used to set the shared library path and run a program. For example, the following runs the gdb debugger on the db_dump utility after setting the appropriate paths:

```
libtool gdb db_dump
```

Libtool may not know what to do with arbitrary commands (it is hardwired to recognize "gdb" and some other commands). If it complains the mode argument will usually resolve the problem:

```
libtool --mode=execute my_debugger db_dump
```

On most systems, using libtool in this way is exactly equivalent to setting the LD_LIBRARY_PATH environment variable and then executing the program. On other systems, using libtool has the virtue of knowing about any other details on systems that don't behave in this typical way.

Running the test suite under UNIX

The Berkeley DB test suite is built if you specify --enable-test as an argument when configuring Berkeley DB. The test suite also requires that you configure and build the Tcl interface to the library.

Before running the tests for the first time, you may need to edit the include.tcl file in your build directory. The Berkeley DB configuration assumes that you intend to use the version of

the tclsh utility included in the Tcl installation with which Berkeley DB was configured to run the test suite, and further assumes that the test suite will be run with the libraries prebuilt in the Berkeley DB build directory. If either of these assumptions are incorrect, you will need to edit the `include.tcl` file and change the following line to correctly specify the full path to the version of tclsh with which you are going to run the test suite:

```
set tclsh_path ...
```

You may also need to change the following line to correctly specify the path from the directory where you are running the test suite to the location of the Berkeley DB Tcl library you built:

```
set test_path ...
```

It may not be necessary that this be a full path if you have configured your system's shared library mechanisms to search the directory where you built or installed the Tcl library.

All Berkeley DB tests are run from within **tclsh**. After starting tclsh, you must source the file `test.tcl` in the test directory. For example, if you built in the `build_unix` directory of the distribution, this would be done using the following command:

```
% source ../test/test.tcl
```

If no errors occur, you should get a "%" prompt.

You are now ready to run tests in the test suite; see Running the test suite (page 488) for more information.

Architecture independent FAQ

1. **I have gcc installed, but configure fails to find it.**

   Berkeley DB defaults to using the native C compiler if none is specified. That is usually "cc", but some platforms require a different compiler to build multithreaded code. To configure Berkeley DB to build with gcc, run configure as follows:

   ```
   env CC=gcc ../dist/configure ...
   ```

2. **When compiling with gcc, I get unreferenced symbols; for example the following:**

   ```
   symbol __muldi3: referenced symbol not found
   symbol __cmpdi2: referenced symbol not found
   ```

   Berkeley DB often uses 64-bit integral types on systems supporting large files, and gcc performs operations on those types by calling library functions. These unreferenced symbol errors are usually caused by linking an application by calling "ld" rather than by calling "gcc": gcc will link in libgcc.a and will resolve the symbols. If that does not help, another possible workaround is to reconfigure Berkeley DB using the --disable-largefile configuration option and then rebuild.

3. **My C++ program traps during a failure in a DB call on my gcc-based system.**

We believe there are some severe bugs in the implementation of exceptions for some gcc compilers. Exceptions require some interaction between compiler, assembler, and runtime libraries. We're not sure exactly what is at fault, but one failing combination is gcc 2.7.2.3 running on SuSE Linux 6.0. The problem on this system can be seen with a rather simple test case of an exception thrown from a shared library and caught in the main program.

A variation of this problem seems to occur on AIX, although we believe it does not necessarily involve shared libraries on that platform.

If you see a trap that occurs when an exception might be thrown by the Berkeley DB runtime, we suggest that you use static libraries instead of shared libraries. See the documentation for configuration. If this doesn't work and you have a choice of compilers, try using a more recent gcc- or a non-gcc based compiler to build Berkeley DB.

Finally, you can disable the use of exceptions in the C++ runtime for Berkeley DB by using the DB_CXX_NO_EXCEPTIONS flag with the DbEnv or Db constructors. When this flag is on, all C++ methods fail by returning an error code rather than throwing an exception.

4. **I get unexpected results and database corruption when running threaded programs.**

   I get error messages that mutex (for example, pthread_mutex_XXX or mutex_XXX) functions are undefined when linking applications with Berkeley DB.

   On some architectures, the Berkeley DB library uses the ISO POSIX standard pthreads and UNIX International (UI) threads interfaces for underlying mutex support; for example, Solaris and HP-UX. You can specify compilers or compiler flags, or link with the appropriate thread library when loading your application to resolve the undefined references:

   ```
   cc ... -lpthread ...
   cc ... -lthread ...
   xlc_r ...
   cc ... -mt ...
   ```

   See the appropriate architecture-specific Reference Guide pages for more information.

   On systems where more than one type of mutex is available, it may be necessary for applications to use the same threads package from which Berkeley DB draws its mutexes. For example, if Berkeley DB was built to use the POSIX pthreads mutex calls for mutex support, the application may need to be written to use the POSIX pthreads interfaces for its threading model. This is only conjecture at this time, and although we know of no systems that actually have this requirement, it's not unlikely that some exist.

   In a few cases, Berkeley DB can be configured to use specific underlying mutex interfaces. You can use the --enable-posixmutexes and --enable-uimutexes configuration options to specify the POSIX and Unix International (UI) threads packages. This should not, however, be necessary in most cases.

   In some cases, it is vitally important to make sure that you load the correct library. For example, on Solaris systems, there are POSIX pthread interfaces in the C library, so applications can link Berkeley DB using only C library and not see any undefined symbols. However, the C library POSIX pthread mutex support is insufficient for Berkeley DB, and

Berkeley DB cannot detect that fact. Similar errors can arise when applications (for example, tclsh) use dlopen to dynamically load Berkeley DB as a library.

If you are seeing problems in this area after you confirm that you're linking with the correct libraries, there are two other things you can try. First, if your platform supports interlibrary dependencies, we recommend that you change the Berkeley DB Makefile to specify the appropriate threads library when creating the Berkeley DB shared library, as an interlibrary dependency. Second, if your application is using dlopen to dynamically load Berkeley DB, specify the appropriate thread library on the link line when you load the application itself.

5. **I get core dumps when running programs that fork children.**

   Berkeley DB handles should not be shared across process forks, each forked child should acquire its own Berkeley DB handles.

6. **I get reports of uninitialized memory reads and writes when running software analysis tools (for example, Rational Software Corp.'s Purify tool).**

   For performance reasons, Berkeley DB does not write the unused portions of database pages or fill in unused structure fields. To turn off these errors when running software analysis tools, build with the --enable-umrw configuration option.

7. **Berkeley DB programs or the test suite fail unexpectedly.**

   The Berkeley DB architecture does not support placing the shared memory regions on remote filesystems -- for example, the Network File System (NFS) or the Andrew File System (AFS). For this reason, the shared memory regions (normally located in the database home directory) must reside on a local filesystem. See Shared memory regions (page 131) for more information.

   With respect to running the test suite, always check to make sure that TESTDIR is not on a remote mounted filesystem.

8. **The db_dump utility fails to build.**

   The db_dump185 utility is the utility that supports the conversion of Berkeley DB 1.85 and earlier databases to current database formats. If the build errors look something like the following, it means the db.h include file being loaded is not a Berkeley DB 1.85 version include file:

```
db_dump185.c: In function `main':
db_dump185.c:210: warning: assignment makes pointer from integer without a cast
db_dump185.c:212: warning: assignment makes pointer from integer without a cast
db_dump185.c:227: structure has no member named `seq'
db_dump185.c:227: `R_NEXT' undeclared (first use in this function)
```

   If the build errors look something like the following, it means that the Berkeley DB 1.85 code was not found in the standard libraries:

```
cc -o db_dump185 db_dump185.o
ld:
Unresolved:
dbopen
```

To build the db_dump185 utility, the Berkeley DB version 1.85 code must already been built and available on the system. If the Berkeley DB 1.85 header file is not found in a standard place, or if the library is not part of the standard libraries used for loading, you will need to edit your Makefile, and change the following lines:

```
DB185INC=
DB185LIB=
```

So that the system Berkeley DB 1.85 header file and library are found; for example:

```
DB185INC=/usr/local/include
DB185LIB=-ldb185
```

AIX

1. **I can't compile and run multithreaded applications.**

   Special compile-time flags are required when compiling threaded applications on AIX. If you are compiling a threaded application, you must compile with the _THREAD_SAFE flag and load with specific libraries; for example, "-lc_r". Specifying the compiler name with a trailing "_r" usually performs the right actions for the system.

   ```
   xlc_r ...
   cc -D_THREAD_SAFE -lc_r ...
   ```

   The Berkeley DB library will automatically build with the correct options.

2. **I can't run using the DB_SYSTEM_MEM option to DB_ENV->open().**

   AIX 4.1 allows applications to map only 10 system shared memory segments. In AIX 4.3, this has been raised to 256K segments, but only if you set the environment variable "export EXTSHM=ON".

3. **On AIX 4.3.2 (or before) I see duplicate symbol warnings when building the C++ shared library and when linking applications.**

   We are aware of some duplicate symbol warnings with this platform, but they do not appear to affect the correct operation of applications.

4. **On AIX 4.3.3 I see undefined symbols for DbEnv::set_error_stream, Db::set_error_stream or DbEnv::verify when linking C++ applications. (These undefined symbols also appear when building the Berkeley DB C++ example applications).**

   By default, Berkeley DB is built with _LARGE_FILES set to 1 to support the creation of "large" database files. However, this also affects how standard classes, like iostream, are named internally. When building your application, use a "-D_LARGE_FILES=1" compilation option, or insert "#define _LARGE_FILES 1" before any #include statements.

5. **I can't create database files larger than 1GB on AIX.**

If you're running on AIX 4.1 or earlier, try changing the source code for os/os_open.c to always specify the **O_LARGEFILE** flag to the open(2) system call, and recompile Berkeley DB from scratch.

Also, the documentation for the IBM Visual Age compiler states that it does not not support the 64-bit filesystem APIs necessary for creating large files; the ibmcxx product must be used instead. We have not heard whether the GNU gcc compiler supports the 64-bit APIs or not.

Finally, to create large files under AIX, the filesystem has to be configured to support large files and the system wide user hard-limit for file sizes has to be greater than 1GB.

6. **I see errors about "open64" when building Berkeley DB applications.**

   System include files (most commonly fcntl.h) in some releases of AIX, HP-UX and Solaris redefine "open" when large-file support is enabled for applications. This causes problems when compiling applications because "open" is a method in the Berkeley DB APIs. To work around this problem:

   a. Avoid including the problematical system include files in source code files which also include Berkeley DB include files and call into the Berkeley DB API.

   b. Before building Berkeley DB, modify the generated include file db.h to itself include the problematical system include files.

   c. Turn off Berkeley DB large-file support by specifying the --disable-largefile configuration option and rebuilding.

FreeBSD

1. **I can't compile and run multithreaded applications.**

   Special compile-time flags are required when compiling threaded applications on FreeBSD. If you are compiling a threaded application, you must compile with the _THREAD_SAFE and -pthread flags:

   ```
   cc -D_THREAD_SAFE -pthread ...
   ```

   The Berkeley DB library will automatically build with the correct options.

2. **I see fsync and close system call failures when accessing databases or log files on NFS-mounted filesystems.**

   Some FreeBSD releases are known to return ENOLCK from fsync and close calls on NFS-mounted filesystems, even though the call has succeeded. The Berkeley DB code should be modified to ignore ENOLCK errors, or no Berkeley DB files should be placed on NFS-mounted filesystems on these systems.

HP-UX

1. **I can't specify the DB_SYSTEM_MEM flag to DB_ENV->open().**

The shmget(2) interfaces are not always used on HP-UX, even though they exist, because anonymous memory allocated using shmget(2) cannot be used to store the standard HP-UX msemaphore semaphores. For this reason, it may not be possible to specify the DB_SYSTEM_MEM flag on some versions of HP-UX. (We have seen this problem only on HP-UX 10.XX, so the simplest workaround may be to upgrade your HP-UX release.)

2. **I can't specify both the DB_PRIVATE and DB_THREAD flags to DB_ENV->open().**

   It is not possible to store the standard HP-UX msemaphore semaphores in memory returned by malloc(3) in some versions of HP-UX. For this reason, it may not be possible to specify both the DB_PRIVATE and DB_THREAD flags on some versions of HP-UX. (We have seen this problem only on some older HP-UX platforms, so the simplest workaround may be to upgrade your HP-UX release.)

3. **I can't compile and run multithreaded applications.**

   Special compile-time flags are required when compiling threaded applications on HP-UX. If you are compiling a threaded application, you must compile with the _REENTRANT flag:

   ```
   cc -D_REENTRANT ...
   ```

   The Berkeley DB library will automatically build with the correct options.

4. **An ENOMEM error is returned from DB_ENV->open() or DB_ENV->remove().**

   Due to the constraints of the PA-RISC memory architecture, HP-UX does not allow a process to map a file into its address space multiple times. For this reason, each Berkeley DB environment may be opened only once by a process on HP-UX; that is, calls to DB_ENV->open() will fail if the specified Berkeley DB environment has been opened and not subsequently closed.

5. **When compiling with gcc, I see the following error:**

   ```
   #error "Large Files (ILP32) not supported in strict ANSI mode."
   ```

   We believe this is an error in the HP-UX include files, but we don't really understand it. The only workaround we have found is to add -D__STDC_EXT__ to the C preprocessor defines as part of compilation.

6. **When using the Tcl or Perl APIs (including running the test suite), I see the error "Can't shl_load() a library containing Thread Local Storage".**

   This problem happens when HP-UX has been configured to use pthread mutex locking, and an attempt is made to call Berkeley DB using the Tcl or Perl APIs. We have never found any way to fix this problem as part of the Berkeley DB build process. To work around the problem, rebuild tclsh or Perl, and modify its build process to explicitly link it against the HP-UX pthread library (currently /usr/lib/libpthread.a).

7. **When running an executable that has been dynamically linked against the Berkeley DB library, I see the error "Can't find path for shared library" even though I correctly set the SHLIB_PATH environment variable.**

By default, some versions of HP-UX ignore the dynamic library search path specified by the SHLIB_PATH environment variable. To work around this, specify the "+s" flag to ld when linking, or run the following command on the executable that is not working:

```
chatr +s enable -l /full/path/to/libdb-3.2.sl ...
```

8. **When building for an IA64 processor, I see either bus errors or compiler warnings about converting between unaligned types (#4232). How can I resolve them?**

   Berkeley DB requires that data types containing members with different sizes be aligned in a consistent way. The HP-UX compiler does not provide this alignment property by default.

   The compiler can be made to generate adequately aligned data by passing the +u1 option to the compiler. See the HP documentation about the +u1 flag [http://docs.hp.com/en/10946/options.htm#opt+u] for more information.

9. **I see errors about "open64" when building Berkeley DB applications.**

   System include files (most commonly fcntl.h) in some releases of AIX, HP-UX and Solaris redefine "open" when large-file support is enabled for applications. This causes problems when compiling applications because "open" is a method in the Berkeley DB APIs. To work around this problem:

   a. Avoid including the problematical system include files in source code files which also include Berkeley DB include files and call into the Berkeley DB API.

   b. Before building Berkeley DB, modify the generated include file db.h to itself include the problematical system include files.

   c. Turn off Berkeley DB large-file support by specifying the --disable-largefile configuration option and rebuilding.

IRIX

1. **I can't compile and run multithreaded applications.**

   Special compile-time flags are required when compiling threaded applications on IRIX. If you are compiling a threaded application, you must compile with the _SGI_MP_SOURCE flag:

   ```
   cc -D_SGI_MP_SOURCE ...
   ```

   The Berkeley DB library will automatically build with the correct options.

Linux

1. **I can't compile and run multithreaded applications.**

   Special compile-time flags are required when compiling threaded applications on Linux. If you are compiling a threaded application, you must compile with the _REENTRANT flag:

   ```
   cc -D_REENTRANT ...
   ```

The Berkeley DB library will automatically build with the correct options.

2. **I see database corruption when accessing databases.**

   Some Linux filesystems do not support POSIX filesystem semantics. Specifically, ext2 and early releases of ReiserFS, and ext3 in some configurations, do not support "ordered data mode" and may insert random data into database or log files when systems crash. Berkeley DB files should not be placed on a filesystem that does not support, or is not configured to support, POSIX semantics.

3. **What scheduler should I use?**

   In some Linux kernels you can select schedulers, and the default is the "anticipatory" scheduler. We recommend not using the "anticipatory" scheduler for transaction processing workloads.

Mac OS X

1. **When trying to link multiple Berkeley DB language interfaces (for example, Tcl, C++, Java, Python) into a single process, I get "multiple definitions" errors from dyld.**

   To fix this problem, set the environment variable MACOSX_DEPLOYMENT_TARGET to 10.3 (or your current version of OS X), and reconfigure and rebuild Berkeley DB from scratch. See the OS X ld(1) and dyld(1) man pages for information about how OS X handles symbol namespaces, as well as undefined and multiply-defined symbols.

2. **When trying to use system-backed shared memory on OS X I see failures about "too many open files".**

   The default number of shared memory segments on OS X is too low. To fix this problem, edit the file /etc/rc, changing the kern.sysv.shmmax and kern.sysv.shmseg values as follows:

```
*** /etc/rc.orig        Fri Dec 19 09:34:09 2003
--- /etc/rc     Fri Dec 19 09:33:53 2003
***************
*** 84,93 ****
   # System tuning
   sysctl -w kern.maxvnodes=$(echo $(sysctl -n hw.physmem) '33554432 /
512 * 1024 +p'|dc)
! sysctl -w kern.sysv.shmmax=4194304
   sysctl -w kern.sysv.shmmin=1
   sysctl -w kern.sysv.shmmni=32
! sysctl -w kern.sysv.shmseg=8
   sysctl -w kern.sysv.shmall=1024
   if [ -f /etc/sysctl-macosxserver.conf ]; then
         awk '{ if (!-1 && -1) print $1 }' <
/etc/sysctl-macosxserver.conf | while read
--- 84,93 ----
   # System tuning
   sysctl -w kern.maxvnodes=$(echo $(sysctl -n hw.physmem) '33554432 /
```

```
512 * 1024 +p'|dc)
! sysctl -w kern.sysv.shmmax=134217728
   sysctl -w kern.sysv.shmmin=1
   sysctl -w kern.sysv.shmmni=32
! sysctl -w kern.sysv.shmseg=32
   sysctl -w kern.sysv.shmall=1024
   if [ -f /etc/sysctl-macosxserver.conf ]; then
         awk '{ if (!-1 && -1) print $1 }' <
 /etc/sysctl-macosxserver.conf | while read
```

and then reboot the system.

OSF/1

1. **I can't compile and run multithreaded applications.**

   Special compile-time flags are required when compiling threaded applications on OSF/1. If you are compiling a threaded application, you must compile with the _REENTRANT flag:

   ```
   cc -D_REENTRANT ...
   ```

   The Berkeley DB library will automatically build with the correct options.

QNX

1. **To what versions of QNX has DB been ported?**

   Berkeley DB has been ported to the QNX Neutrino technology which is commonly referred to as QNX RTP (Real-Time Platform). Berkeley DB has not been ported to earlier versions of QNX, such as QNX 4.25.

2. **Building Berkeley DB shared libraries fails.**

   The `/bin/sh` utility distributed with some QNX releases drops core when running the GNU libtool script (which is used to build Berkeley DB shared libraries). There are two workarounds for this problem: First, only build static libraries. You can disable building shared libraries by specifying the configuration flag when configuring Berkeley DB.

   Second, build Berkeley DB using an alternate shell. QNX distributions include an accessories disk with additional tools. One of the included tools is the GNU bash shell, which is able to run the libtool script. To build Berkeley DB using an alternate shell, move `/bin/sh` aside, link or copy the alternate shell into that location, configure, build and install Berkeley DB, and then replace the original shell utility.

3. **Are there any QNX filesystem issues?**

   Berkeley DB generates temporary files for use in transactionally protected file system operations. Due to the filename length limit of 48 characters in the QNX filesystem, applications that are using transactions should specify a database name that is at most 43 characters.

4. **What are the implications of QNX's requirement to use shm_open(2) in order to use mmap(2)?**

   QNX requires that files mapped with mmap(2) be opened using shm_open(2). There are other places in addition to the environment shared memory regions, where Berkeley DB tries to memory map files if it can.

   The memory pool subsystem normally attempts to use mmap(2) even when using private memory, as indicated by the DB_PRIVATE flag to DB_ENV->open(). In the case of QNX, if an application is using private memory, Berkeley DB will not attempt to map the memory and will instead use the local cache.

5. **What are the implications of QNX's mutex implementation using microkernel resources?**

   On QNX, the primitives implementing mutexes consume system resources. Therefore, if an application unexpectedly fails, those resources could leak. Berkeley DB solves this problem by always allocating mutexes in the persistent shared memory regions. Then, if an application fails, running recovery or explicitly removing the database environment by calling the DB_ENV->remove() method will allow Berkeley DB to release those previously held mutex resources. If an application specifies the DB_PRIVATE flag (choosing not to use persistent shared memory), and then fails, mutexes allocated in that private memory may leak their underlying system resources. Therefore, the DB_PRIVATE flag should be used with caution on QNX.

SCO

1. **If I build with gcc, programs such as db_dump and db_stat core dump immediately when invoked.**

   We suspect gcc or the runtime loader may have a bug, but we haven't tracked it down. If you want to use gcc, we suggest building static libraries.

Solaris

1. **I can't compile and run multithreaded applications.**

   Special compile-time flags and additional libraries are required when compiling threaded applications on Solaris. If you are compiling a threaded application, you must compile with the D_REENTRANT flag and link with the libpthread.a or libthread.a libraries:

   ```
   cc -mt ...
   cc -D_REENTRANT ... -lthread
   cc -D_REENTRANT ... -lpthread
   ```

   The Berkeley DB library will automatically build with the correct options.

2. **I've installed gcc on my Solaris system, but configuration fails because the compiler doesn't work.**

   On some versions of Solaris, there is a cc executable in the user's path, but all it does is display an error message and fail:

```
% which cc
/usr/ucb/cc
% cc
/usr/ucb/cc: language optional software package not installed
```

Because Berkeley DB always uses the native compiler in preference to gcc, this is a fatal error. If the error message you are seeing is the following, then this may be the problem:

```
checking whether the C compiler (cc -O) works... no
configure: error: installation or configuration problem: C compiler cannot create executabl
```

The simplest workaround is to set your CC environment variable to the system compiler and reconfigure; for example:

```
env CC=gcc ../dist/configure
```

If you are using the --configure-cxx option, you may also want to specify a C++ compiler, for example the following:

```
env CC=gcc CCC=g++ ../dist/configure
```

3. **I see the error "libc internal error: _rmutex_unlock: rmutex not held", followed by a core dump when running threaded or JAVA programs.**

   This is a known bug in Solaris 2.5 and it is fixed by Sun patch 103187-25.

4. **I see error reports of nonexistent files, corrupted metadata pages and core dumps.**

   Solaris 7 contains a bug in the threading libraries (-lpthread, -lthread), which causes the wrong version of the pwrite routine to be linked into the application if the thread library is linked in after the C library. The result will be that the pwrite function is called rather than the pwrite64. To work around the problem, use an explicit link order when creating your application.

   Sun Microsystems is tracking this problem with Bug Id's 4291109 and 4267207, and patch 106980-09 to Solaris 7 fixes the problem:

   ```
   Bug Id: 4291109
   Duplicate of: 4267207
   Category: library
   Subcategory: libthread
   State: closed
   Synopsis: pwrite64 mapped to pwrite
   Description:
   When libthread is linked after libc, there is a table of functions in
   libthread that gets "wired into" libc via _libc_threads_interface().
   The table in libthread is wrong in both Solaris 7 and on28_35 for the
   TI_PWRITE64 row (see near the end).
   ```

5. **I see corrupted databases when doing hot backups or creating a hot failover archive.**

The Solaris cp utility is implemented using the mmap system call, and so writes are not blocked when it reads database pages. See Berkeley DB recoverability (page 180) for more information.

6. **Performance is slow and the application is doing a lot of I/O to the disk on which the database environment's files are stored.**

   By default, Solaris periodically flushes dirty blocks from memory-mapped files to the backing filesystem. This includes the Berkeley DB database environment's shared memory regions and can affect Berkeley DB performance. Workarounds include creating the shared regions in system shared memory (DB_SYSTEM_MEM) or application private memory (DB_PRIVATE), or configuring Solaris to not flush memory-mapped pages. For more information, see the "Solaris Tunable Parameters Reference Manual: fsflush and Related Tunables".

7. **I see errors about "open64" when building Berkeley DB applications.**

   System include files (most commonly fcntl.h) in some releases of AIX, HP-UX and Solaris redefine "open" when large-file support is enabled for applications. This causes problems when compiling applications because "open" is a method in the Berkeley DB APIs. To work around this problem:

   a. Avoid including the problematical system include files in source code files which also include Berkeley DB include files and call into the Berkeley DB API.

   b. Before building Berkeley DB, modify the generated include file db.h to itself include the problematical system include files.

   c. Turn off Berkeley DB large-file support by specifying the --disable-largefile configuration option and rebuilding.

SunOS

1. **I can't specify the DB_SYSTEM_MEM flag to DB_ENV->open().**

   The shmget(2) interfaces are not used on SunOS releases prior to 5.0, even though they apparently exist, because the distributed include files did not allow them to be compiled. For this reason, it will not be possible to specify the DB_SYSTEM_MEM flag to those versions of SunOS.

Ultrix

1. **Configuration complains that mmap(2) interfaces aren't being used.**

   The mmap(2) interfaces are not used on Ultrix, even though they exist, because they are known to not work correctly.

Chapter 28. Building Berkeley DB for Windows

This chapter contains general instructions on building Berkeley DB for specific windows platforms using specific compilers. The Windows FAQ (page 320) also contains helpful information.

The `build_windows` directory in the Berkeley DB distribution contains project files for Microsoft Visual Studio:

Project File	Description
Berkeley_DB.sln	Visual Studio 2005 (8.0) workspace
*.vcproj	Visual Studio 2005 (8.0) projects
Berkeley_DB.dsw	Visual C++ 6.0 workspace
*.dsp	Visual C++ 6.0 projects

These project files can be used to build Berkeley DB for the following platforms: Windows NT/2K/XP/2003/Vista, and 64-bit Windows XP/2003/Vista.

Building Berkeley DB for 32 bit Windows

Visual C++ .NET 2008

1. Choose *File* -> *Open* -> *Project/Solution...*. In the `build_windows` directory, select `Berkeley_DB.sln` and click Open.
2. The *Visual Studio Conversion Wizard* will open automatically. Click the *Finish* button.
3. On the next screen click the *Close* button.
4. Choose the desired project configuration from the drop-down menu on the tool bar (either Debug or Release).
5. Choose the desired platform configuration from the drop-down menu on the tool bar (usually Win32 or x64).
6. To build, right-click on the Berkeley_DB solution and select Build Solution.

Visual C++ .NET 2005

1. Choose *File* -> *Open* -> *Project/Solution...*. In the `build_windows` directory, select `Berkeley_DB.sln` and click Open
2. Choose the desired project configuration from the drop-down menu on the tool bar (either Debug or Release).
3. Choose the desired platform configuration from the drop-down menu on the tool bar (usually Win32 or x64).
4. To build, right-click on the Berkeley_DB solution and select Build Solution.

Visual C++ .NET or Visual C++ .NET 2003

NOTE: Although Visual Studio .NET uses solution files (.sln), the solution file shipped with Berkeley DB is not compatible with Visual Studio .NET. You need to upgrade from the Visual C++ 6.0 workspace to build with Visual C++ .NET or Visual C++ .NET 2003.

1. Choose *File -> Open Solution*. Look in the `build_windows` directory for workspace files, select `Berkeley_DB.dsw`, and press Open.

2. You will be prompted to convert the project files to current Visual C++ format. Select "Yes to All".

3. Choose the project configuration from the drop-down menu on the .NET tool bar (Debug or Release).

4. To build, right-click on build_all and select Build.

Visual C++ 6.0

1. Choose *File -> Open Workspace*. Look in the `build_windows` directory for Workspaces, select `Berkeley_DB.dsw`, and press Open.

2. Choose the desired project configuration by going to *Build -> Set Active Configuration* and select the appropriate option to the build_all project (Debug or Release). Then click OK.

3. To build, press F7.

Build results

The results of your build will be placed in one of the following Berkeley DB subdirectories, depending on the configuration that you chose:

```
build_windows\Win32\Debug
build_windows\Win32\Release
build_windows\Win32\Debug_static
build_windows\Win32\Release_static
```

When building your application during development, you should normally use compile options "Debug Multithreaded DLL" and link against `build_windows\Debug\libdb48d.lib`. You can also build using a release version of the Berkeley DB libraries and tools, which will be placed in `build_windows\Win32\Release\libdb48.lib`. When linking against the release build, you should compile your code with the "Release Multithreaded DLL" compile option. You will also need to add the `build_windows` directory to the list of include directories of your application's project, or copy the Berkeley DB include files to another location.

Building Berkeley DB for 64-bit Windows

The following procedure can be used to build natively on a 64-bit system or to cross-compile from a 32-bit system.

When building 64-bit binaries, the output directory will be one of the following Berkeley DB subdirectories, depending upon the configuration that you chose:

```
build_windows\x64\Debug
build_windows\x64\Release
build_windows\x64\Debug_static
build_windows\x64\Release_static
```

x64 build with Visual Studio 2005 or newer

1. Follow the build instructions for your version of Visual Studio, as described in Building Berkeley DB for 32 bit Windows (page 311).

2. Select *x64* from the *Platform Configuration* dropdown.

3. Right click on *Solution 'Berkeley_DB'* in the solution explorer, and select *Build Solution*

x64 build with Visual Studio .NET 2003 or earlier

The build files shipped for earlier version of Visual Studio are not configured to build 64-bit binary files. It is possible to do so, but we recommend upgrading to a newer version of Visual Studio.

Building Berkeley DB with Cygwin

To build Berkeley DB with Cygwin, follow the instructions in Building for UNIX/POSIX (page 289).

Building the C++ API

C++ support is built automatically on Windows.

Building the C++ STL API

Note that MS's compilers older than that of MS Visual C++ 2005 are not supported by or tested with the STL API. The oldest MS compiler/IDE you can use to build the STL API is MS Visual C++ 2005.

In the project list of the `Berkeley_DB.sln` solution, build the "db_stl" project and "db_stl_static" project to build STL API as a dynamic or static library respectively. And in your application, you should link this library file as well as the Berkeley DB library file to your application. The STL API library file is by default always put at the same location as the Berkeley DB library file.

And you need to include the STL API header files in your application code. If you are using the Berkeley DB source tree, the header files are in <Berkeley DB Source Root >/stl directory; If you are using the pre-built installed version, these header files are in < Berkeley DB Installed Directory>/include, as well as the db.h and db_cxx.h header files.

Building the Java API

Java support is not built automatically. The following instructions assume that you have installed the Sun Java Development Kit in `d:\java`. Of course, if you installed elsewhere or have different Java software, you will need to adjust the pathnames accordingly.

Building Java with Visual C++ .NET or above

1. Set your include directories. Choose *Tools -> Options -> Projects -> VC++ Directories*. Under the "Show directories for" pull-down, select "Include files". Add the full pathnames for the `d:\java\include` and `d:\java\include\win32` directories. Then click OK. These are the directories needed when including jni.h.

2. Set the executable files directories. Choose *Tools -> Options -> Projects -> VC++ Directories*. Under the "Show directories for" pull-down, select "Executable files". Add the full pathname for the `d:\java\bin` directory, then click OK. This is the directory needed to find javac.

3. Set the build type to Release or Debug in the drop-down on the tool bar.

4. To build, right-click on db_java and select Build. This builds the Java support library for Berkeley DB and compiles all the java files, placing the resulting `db.jar` and `dbexamples.jar` files in one of the following Berkeley DB subdirectories, depending on the configuration that you chose:

   ```
   build_windows\Win32\Debug
   build_windows\Win32\Release
   ```

Building Java with Visual C++ 6.0

1. Set the include directories. Choose *Tools -> Options -> Directories*. Under the "Show directories for" pull-down, select "Include files". Add the full pathnames for the `d:\java\include` and `d:\java\include\win32` directories. These are the directories needed when including `jni.h`.

2. Set the executable files directories. Choose *Tools -> Options -> Directories*. Under the "Show directories for" pull-down, select "Executable files". Add the full pathname for the `d:\java\bin` directory. This is the directory needed to find javac.

3. Go to *Build -> Set Active Configuration* and select either the Debug or Release version of the db_java project. Then press OK.

4. To build, select *Build -> Build libdb_java48.dll*. This builds the Java support library for Berkeley DB and compiles all the java files, placing the resulting `db.jar` and `dbexamples.jar` files in one of the following Berkeley DB subdirectories, depending on the configuration that you chose:

   ```
   build_windows\Win32\Debug
   build_windows\Win32\Release
   ```

To run Java code, set your environment variable CLASSPATH to include the full pathname of these jar files, and your environment variable PATH to include the build_windows\Win32\Release subdirectory. On Windows, remember that files or directories in the CLASSPATH and PATH variables must be separated by semicolons (unlike UNIX). Then, try running the following command as a test:

```
java db.AccessExample
```

If you want to run Java code using a Debug build, substitute 'Debug' for 'Release' in the instructions above. Make sure you use the Debug JAR file with the Debug DLL and the Release JAR with the Release DLL.

Building the C# API

The C# support is built by a separate Visual Studio solution, build_windows\BDB_dotnet.sln, and requires version 2.0 (or higher) of the .NET platform.

Building C# with Visual Studio 2005

By default, the solution will build the native libraries, the managed assembly and all example programs. The NUnit tests need to be built explicitly because of their dependence upon the NUnit assembly. The native libraries will be placed in one of the following subdirectories, depending upon the chosen configuration:

```
build_windows\Win32\Debug
build_windows\Win32\Release
build_windows\x64\Debug
build_windows\x64\Release
```

The managed assembly and all C# example programs will be placed in one of the following subdirectories, depending upon the chosen configuration:

```
build_windows\AnyCPU\Debug
build_windows\AnyCPU\Release
```

The native libraries need to be locatable by the .NET platform, meaning they must be copied into an application's directory, the Windows or System directory, or their location must be added to the PATH environment variable. The example programs demonstrate how to programmatically edit the PATH variable.

Building the Tcl API

Tcl support is not built automatically. See Loading Berkeley DB with Tcl (page 266) for information on sites from which you can download Tcl and which Tcl versions are compatible with Berkeley DB. These notes assume that Tcl is installed as d:\tcl, but you can change that if you want.

The Tcl library must be built as the same build type as the Berkeley DB library (both Release or both Debug). We found that the binary release of Tcl can be used with the Release configuration of Berkeley DB, but you will need to build Tcl from sources for the Debug configuration. Before building Tcl, you will need to modify its makefile to make sure that you

are building a debug version, including thread support. This is because the set of DLLs linked into the Tcl executable must match the corresponding set of DLLs used by Berkeley DB.

Building Tcl with Visual C++ .NET or above

1. Set the include directories. Choose *Tools -> Options -> Projects -> VC++ Directories*. Under the "Show directories for" pull-down, select "Include files". Add the full pathname for `d:\tcl\include`, then click OK. This is the directory that contains `tcl.h`.

2. Set the library files directory. Choose *Tools -> Options -> Projects -> VC++ Directories*. Under the "Show directories for" pull-down, select "Library files". Add the full pathname for the `d:\tcl\lib` directory, then click OK. This is the directory needed to find `tcl84g.lib` (or whatever the library is named in your distribution).

3. Set the build type to Release or Debug in the drop-down on the tool bar.

4. To build, right-click on db_tcl and select Build. This builds the Tcl support library for Berkeley DB, placing the result into one of the following Berkeley DB subdirectories, depending upon the configuration that you chose:

   ```
   build_windows\Win32\Debug\libdb_tcl48d.dll
   build_windows\Win32\Release\libdb_tcl48.dll
   ```

If you use a version different from Tcl 8.4.x you will need to change the name of the Tcl library used in the build (for example, `tcl84g.lib`) to the appropriate name. To do this, right click on *db_tcl*, go to *Properties -> Linker -> Input -> Additional dependencies* and change `tcl84g.lib` to match the Tcl version you are using.

Building Tcl with Visual C++ 6.0

1. Set the include directories. Choose *Tools -> Options -> Directories*. Under the "Show directories for" pull-down, select "Include files". Add the full pathname for `d:\tcl\include`, then click OK. This is the directory that contains `tcl.h`.

2. Set the library files directory. Choose *Tools -> Options -> Directories*. Under the "Show directories for" pull-down, select "Library files". Add the full pathname for the `d:\tcl\lib` directory, then click OK. This is the directory needed to find `tcl84g.lib` (or whatever the library is named in your distribution).

3. Go to *Build -> Set Active Configuration* and select either the Debug or Release version of the db_tcl project. Then press OK.

4. To build, select *Build -> Build libdb_tcl48.dll*. This builds the Tcl support library for Berkeley DB, placing the result into one of the following Berkeley DB subdirectories, depending upon the configuration that you chose:

   ```
   build_windows\Win32\Debug\libdb_tcl48d.dll
   build_windows\Win32\Release\libdb_tcl48.dll
   ```

If you use a version different from Tcl 8.4.x you will need to change the name of the Tcl library used in the build (for example, `tcl84g.lib`) to the appropriate name. To do this, choose *Project -> Settings -> db_tcl* and change the Tcl library listed in the Object/Library modules `tcl84g.lib` to match the Tcl version you are using.

Distributing DLLs

When distributing applications linked against the DLL (not static) version of the library, the DLL files you need will be found in one of the following Berkeley DB subdirectories, depending upon the configuration that you chose:

```
build_windows\Win32\Debug
build_windows\Win32\Release
build_windows\Win32\Debug_static
build_windows\Win32\Release_static
build_windows\x64\Debug
build_windows\x64\Release
build_windows\x64\Debug_static
build_windows\x64\Release_static
```

You may also need to redistribute DLL files needed for the compiler's runtime. For Visual C++ 6.0, these files are `msvcrt.dll` and `msvcp60.dll` if you built with a Release configuration, or `msvcrtd.dll` and `msvcp60d.dll` if you are using a Debug configuration. Generally, these runtime DLL files can be installed in the same directory that will contain your installed Berkeley DB DLLs. This directory may need to be added to your System PATH environment variable. Check your compiler's license and documentation for specifics on redistributing runtime DLLs.

Building a small memory footprint library

For applications that don't require all of the functionality of the full Berkeley DB library, an option is provided to build a static library with certain functionality disabled. In particular, cryptography, hash and queue access methods, replication and verification are all turned off. This can reduce the memory footprint of Berkeley DB significantly.

In general on Windows systems, you will want to evaluate the size of the final application, not the library build. The Microsoft LIB file format (like UNIX archives) includes copies of all of the object files and additional information. The linker rearranges symbols and strips out the overhead, and the resulting application is much smaller than the library. There is also a Visual C++ optimization to "Minimize size" that will reduce the library size by a few percent.

A Visual C++ project file called `db_small` is provided for this small memory configuration. During a build, static libraries are created in `Release` or `Debug`, respectively. The library name is `libdb_small48sd.lib` for the debug build, or `libdb_small48s.lib` for the release build.

For assistance in further reducing the size of the Berkeley DB library, or in building small memory footprint libraries on other systems, please contact Berkeley DB support.

Running the test suite under Windows

To build the test suite on Windows platforms, you will need to configure Tcl support. You will also need sufficient main memory (at least 64MB), and disk (around 250MB of disk will be sufficient).

Building the software needed by the tests

The test suite must be run against a Debug version of Berkeley DB, so you will need a Debug version of the Tcl libraries. This involves building Tcl from its source. See the Tcl sources for more information. Then build the Tcl API - see Building the Tcl API (page 315) for details.

Visual Studio 2005 or newer

To build for testing, perform the following steps:

1. Open the Berkeley DB solution.
2. Ensure that the target configuration is Debug
3. Right click the *db_tcl* project in the Solution Explorer, and select *Build*.
4. Right click the *db_test* project in the Solution Explorer, and select *Build*.

Visual Studio 2003 .NET or earlier

To build for testing, perform the following steps:

1. Open the Berkeley DB workspace.
2. In Visual C++ 6.0, set the active configuration to db_test -- Debug. To set an active configuration, under the *Build* menu, select *Set Active Configuration*. Then choose *db_test -- Debug*. In Visual C++ .NET, just make sure *Debug* is selected in the drop down list on the tool bar.
3. Build. In Visual C++ 6.0, the IDE menu item for this is called "build dbkill.exe", even though dbkill is just one of the things that is built. In Visual C++ .NET, right-click on the *db_test* project and select *Build*. This step makes sure that the base Berkeley DB .dll, tcl support, and various tools that are needed by the test suite are all built.

Running the test suite under Windows

Before running the tests for the first time, you must edit the file `include.tcl` in your build directory and change the line that reads:

```
set tclsh_path SET_YOUR_TCLSH_PATH
```

You will want to use the location of the `tclsh` program (be sure to include the name of the executable). For example, if Tcl is installed in `d:\tcl`, this line should be the following:

```
set tclsh_path d:\tcl\bin\tclsh84g.exe
```

If your path includes spaces be sure to enclose it in quotes:

```
set tclsh_path "c:\Program Files\tcl\bin\tclsh84g.exe"
```

Make sure that the path to Berkeley DB's tcl library is in your current path. On Windows NT/2000/XP, edit your PATH using the My Computer -> Properties -> Advanced -> Environment Variables dialog. On earlier versions of Windows, you may find it convenient to add a line to c:\AUTOEXEC.BAT:

```
SET PATH=%PATH%;c:\db\build_windows
```

Then, in a shell of your choice enter the following commands:

1. cd build_windows

2. run `d:\tcl\bin\tclsh84g.exe`, or the equivalent name of the Tcl shell for your system.

   You should get a "%" prompt.

3. % source ../test/test.tcl
   If no errors occur, you should get a "%" prompt.

You are now ready to run tests in the test suite; see Running the test suite (page 488) for more information.

Windows notes

If a system memory environment is closed by all processes, subsequent attempts to open it will return an error. To successfully open a transactional environment in this state, recovery must be run by the next process to open the environment. For non-transactional environments, applications should remove the existing environment and then create a new database environment.

1. Berkeley DB does not support the Windows/95, Windows/98 or Windows/ME platforms.

2. On Windows, system paging file memory is freed on last close. For this reason, multiple processes sharing a database environment created using the DB_SYSTEM_MEM flag must arrange for at least one process to always have the environment open, or alternatively that any process joining the environment be prepared to re-create it.

3. When using the DB_SYSTEM_MEM flag, Berkeley DB shared regions are created without ACLs, which means that the regions are only accessible to a single user. If wider sharing is appropriate (for example, both user applications and Windows/NT service applications need to access the Berkeley DB regions), the Berkeley DB code will need to be modified to create the shared regions with the correct ACLs. Alternatively, by not specifying the DB_SYSTEM_MEM flag, filesystem-backed regions will be created instead, and the permissions on those files may be directly specified through the DB_ENV->open() method.

4. Applications that operate on wide character strings can use the Windows function WideCharToMultiByte with the code page CP_UTF8 to convert paths to the form expected by Berkeley DB. Internally, Berkeley DB calls MultiByteToWideChar on paths before calling Windows functions.

5. Various Berkeley DB methods take a **mode** argument, which is intended to specify the underlying file permissions for created files. Berkeley DB currently ignores this argument on Windows systems.

   It would be possible to construct a set of security attributes to pass to **CreateFile** that accurately represents the mode. In the worst case, this would involve looking up user and all group names, and creating an entry for each. Alternatively, we could call the **_chmod** (partial emulation) function after file creation, although this leaves us with an obvious race.

   Practically speaking, however, these efforts would be largely meaningless on a FAT file system, which only has a "readable" and "writable" flag, applying to all users.

Windows FAQ

1. **Why do I have db_load.dll - where is db_load.exe?**

   Microsoft Visual C++ .NET has some bugs related to converting project files from VC++ 6.0 format and incorrectly converts the db_load project. For more information, see Microsoft's article about this bug [http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q321274].

   The workaround is simply to switch the **db_load** project back to generating an EXE after converting to VC++ .NET. To do this, right click on db_load -> Properties and change "Configuration Type" from "Dynamic Library (.dll)" to "Application (.exe)".

2. **My Win* C/C++ application crashes in the Berkeley DB library when Berkeley DB calls fprintf (or some other standard C library function).**

   You should be using the "Debug Multithreaded DLL" compiler option in your application when you link with the build_windows\Debug\libdb48d.lib library (this .lib file is actually a stub for libdb48d.DLL). To check this setting in Visual C++, choose the *Project/Settings* menu item and select *Code Generation* under the tab marked *C/C++*; and see the box marked *Use runtime library*. This should be set to *Debug Multithreaded DLL*. If your application is linked against the static library, build_windows\Debug\libdb48sd.lib; then, you will want to set *Use runtime library* to *Debug Multithreaded*.

   Setting this option incorrectly can cause multiple versions of the standard libraries to be linked into your application (one on behalf of your application, and one on behalf of the Berkeley DB library). That violates assumptions made by these libraries, and traps can result.

3. **Why are the build options for DB_DLL marked as "Use MFC in a Shared DLL"? Does Berkeley DB use MFC?**

   Berkeley DB does not use MFC at all. It does however, call malloc and free and other facilities provided by the Microsoft C runtime library. We found in our work that many applications and libraries are built assuming MFC, and specifying this for Berkeley DB solves various interoperation issues, and guarantees that the right runtime libraries are selected. Note that because we do not use MFC facilities, the MFC library DLL is not marked as a dependency for libdb.dll, but the appropriate Microsoft C runtime is.

4. **How can I build Berkeley DB for MinGW [http://www.mingw.org]?**

Follow the instructions in Building for UNIX/POSIX (page 289), and specify the --enable-mingw option to the configuration script. This configuration option currently only builds static versions of the library, it does not yet build a DLL version of the library, and file sizes are limited to 2GB (2^32 bytes.)

5. **How can I build a Berkeley DB for Windows 98/ME?**

   Windows 98/ME is no longer supported by Berkeley DB. The following is therefore only of interest to historical users of Berkeley DB.

   By default on Windows, Berkeley DB supports internationalized filenames by treating all directory paths and filenames passed to Berkeley DB methods as UTF-8 encoded strings. All paths are internally converted to wide character strings and passed to the wide character variants of Windows system calls.

   This allows applications to create and open databases with names that cannot be represented with ASCII names while maintaining compatibility with applications that work purely with ASCII paths.

   Windows 98 and ME do not support Unicode paths directly. To build for those versions of Windows, either:

   - Follow the instructions at Microsoft's web site [http://msdn.microsoft.com/goglobal/bb688166.aspx].
   - Open the workspace or solution file with Visual Studio. Then open the Project properties/settings section for the project you need to build (at least db_dll). In the *C/C++->Preprocessor->Preprocessor Definitions* section, remove *_UNICODE* and *UNICODE* entries. Add in an entry of *_MBCS*. Build the project as normal.

   The ASCII builds will also work on Windows NT/2K/XP and 2003, but will not translate paths to wide character strings.

Chapter 29. Building Berkeley DB for Windows CE

Building for Windows CE

This page contains general instructions on building Berkeley DB for Windows CE platforms using specific compilers.

The `build_wince` directory in the Berkeley DB distribution contains project files for Microsoft eMbedded Visual C++:

Project File	Description
Berkeley_DB.vcw	eMbedded Visual C++ 4.0 workspace
*.vcp	eMbedded Visual C++ 4.0 projects

These project files can be used to build Berkeley DB for the Windows CE platform.

Building Berkeley DB for Windows CE

eMbedded Visual C++ 4.0

1. Choose *File -> Open Workspace...*. Navigate to the `build_wince` directory, select `Berkeley_DB` and click Open.

2. Choose the project configuration from the *Build -> Set Active Configuration...* drop-down menu on the tool bar. The correct target will usually db_small - Win32 (WCE emulator) Debug or Release.

3. To build, press F7, or select *Build* from the drop-down menu on the tool bar.

Build results

The results of your build will be placed in a subdirectory of `build_windows` named after the configuration you chose (for examples, `build_wince\Release` or `build_wince\Debug`).

When building your application during development, you should normally link against `build_wince\Debug\libdb_small48sd.lib`. You can also build using a release version of the Berkeley DB libraries and tools, which will be placed in `build_windows\Release\libdb_small48s.lib`. You will also need to add the `build_wince` directory to the list of include directories of your application's project, or copy the Berkeley DB include files to a location in your Visual Studio include path.

Building Berkeley DB for different target CPU architectures

There are many possible target CPU architectures for a Windows CE application. This section outlines the process required to add a new target architecture to the project files supplied with Berkeley DB.

eMbedded Visual C++ 4.0

1. Choose *File -> Open Workspace...*. Navigate to the `build_wince` directory, select `Berkeley_DB` and click Open.

2. Choose the *Build -> Configurations...* menu item.

3. Click the *Add...* button.

4. Select the desired CPU architecture from the first dropdown box. Select an existing target to copy the settings from (The corresponding emulator target is a good choice). Configuration should be either Debug or Release. Click *OK*.

5. Choose the *Build -> Set Active Configuration...* menu item. Choose the new target then click OK.

6. Select the *Project -> Settings...* menu item. Under the C/C++ tab, select the Preprocessor *Category*. In the *Additional include directories:* field add: ".,.." without the quotes. Click OK.

7. Build as per the instructions in *Building Berkeley DB for Windows CE* above.

Windows CE notes

1. The C++ API is not supported on Windows CE. The file stream and exception handling functionality provided by the Berkeley DB C++ API are not supported by Windows CE. It is possible to build a C++ application against the Berkeley DB C API.

2. We do not currently ship workspace/project files for Windows CE that are compatible with Visual Studio 2005. You should be able to manually create project files by duplicating the structure in the eMbedded Visual C++ 4.0 projects.

3. The Java API is not currently supported on Windows CE.

4. Tcl support is not currently supported on Windows CE.

5. Berkeley DB is shipped with support for the Pocket PC 2003 and Smartphone 2003 target platforms. It is possible to build Berkeley DB for different target platforms using Visual Studio's Configuration Manager.
   This can be done using the following steps:

   a. Open Visual Studio, and load the `build_wince/Berkeley_DB.sln` solution file.

   b. Select the *Build->Configuration Manager...* menu item.

   c. In the *Active Solution Platform...* dropdown, select *New...*

   d. Select the desired target platform (you must have the desired Microsoft Platform SDK installed for it to appear in the list). Choose to copy settings from either the Pocket PC 2003 or Smartphone 2003 platforms.

Before building the wce_tpcb sample application for the new platform, you will need to complete the following steps:

a. Open the project properties page for wce_tpcb. Do this by: Right click *wce_tpcb* in the *Solution Explorer* then select *Properties*

b. Select *Configuration Properties->Linker->Input*

c. Remove `secchk.lib` and `crtti.lib` from the *Additional Dependencies* field.

NOTE: These steps are based on Visual Studio 2005, and might vary slightly depending on which version of Visual Studio being used.

Windows CE/Mobile FAQ

1. **What if my Windows CE/Mobile device does not support SetFilePointer and/or SetEndOfFile?**

   You can manually disable the truncate functionality from the build.

   Do that by opening the `db-X.X.X/build_wince/db_config.h` file, and change the line that reads

   ```
   #define HAVE_FTRUCATE 1
   ```

   to read

   ```
   #undef HAVE_FTRUNCATE
   ```

   Making this change disables DB->compact() for btree databases.

2. **Why doesn't automatic log archiving work?**

   The Windows CE/Mobile platform does not have a concept of a working directory. This means that the DB_ARCH_REMOVE and DB_ARCH_ABS flags do not work properly within Windows CE, since they rely on having a working directory.

   To work around this issue, you can call log_archive with the DB_ARCH_LOG flag, the list of returned file handles will not contain absolute paths. Your application can take this list of files, construct absolute paths, and delete the files.

3. **Does Berkeley DB support Windows Mobile?**

   Yes.

   Berkeley DB relies on a subset of the Windows API, and some standard C library APIs. These are provided by Windows CE. Windows Mobile is built "on top" of Windows CE.

4. **I see a file mapping error when opening a Berkeley DB environment or database. What is wrong?**

   The default behavior of Berkeley DB is to use memory mapped files in the environment. It seems that Windows CE does not allow memory mapped files to be created on flash storage.

There are two workarounds:

a. Configure the Berkeley DB environment not to use memory mapped files. The options are discussed in detail in the section called "Shared memory regions" (page 131).

b. Create the Berkeley DB environment on non-flash storage. It is possible to store database and log files in a different location to using the DB_ENV->set_data_dir() and DB_ENV->set_lg_dir() APIs.

Chapter 30. Building Berkeley DB for VxWorks

Building for VxWorks 5.4 and 5.5

The build_vxworks directory in the Berkeley DB distribution contains a workspace and project files for Tornado 2.0/VxWorks 5.4 and Tornado 2.2/VxWorks 5.5.

File	Description
BerkeleyDB20.wsp	Berkeley DB Workspace file for Tornado 2.0
BerkeleyDB20.wpj	Berkeley DB Project file for Tornado 2.0
BerkeleyDB22.wsp	Berkeley DB Workspace file for Tornado 2.2
BerkeleyDB22.wpj	Berkeley DB Project file for Tornado 2.2
dbdemo/dbdemo20.wpj	VxWorks notes (page 328) project file for Tornado 2.0
dbdemo/dbdemo22.wpj	VxWorks notes (page 328) project file for Tornado 2.2
db_*/*20.wpj	VxWorks notes (page 328) project files for Tornado 2.0
db_*/*22.wpj	VxWorks notes (page 328) project files for Tornado 2.2

Building With Tornado 2.0 or Tornado 2.2

Open the workspace **BerkeleyDB20.wsp** or **BerkeleyDB22.wsp**. The list of projects in this workspace will be shown. These projects were created for the x86 BSP for VxWorks.

The remainder of this document assumes that you already have a VxWorks target and a target server, both up and running. It also assumes that your VxWorks image is configured properly for your needs. It also assumes that you have an acceptable file system already available. See VxWorks FAQ (page 329) for more information about file system requirements. See VxWorks notes (page 328) for more information about building a small footprint version of Berkeley DB.

First, you need to set the include directories. To do this, go to the *Builds* tab for the workspace. Open up *Berkeley DB Builds*. You will see several different builds, containing different configurations. All of the projects in the Berkeley DB workspace are created to be downloadable applications.

Build	Description
PENTIUM_debug	x86 BSP with debugging
PENTIUM_release	x86 BSP no debugging

You have to add a new build specification if you use a different BSP, want to add a build for the simulator or want to customize further. For instance, if you have the Power PC (PPC) BSP, you need to add a new build for the PPC tool chain. To do so, select the "Builds" tab, select

the Berkeley DB project name, and right-click. Choose the *New Build...* selection and create the new build target. For your new build target, you need to decide whether it should be built for debugging. See the properties of the Pentium builds for ways to configure for each case. After you add this build you, you still need to configure correctly the include directories, as described in the sections that follow.

If you are running with a different BSP, you should remove the build specifications that do not apply to your hardware. We recommend that you do this after you configure any new build specifications first. The Tornado tools will get confused if you have a PENTIUMgnu build specification for a PPC BSP, for instance.

Select the build you are interested in, and right-click. Choose the *Properties...* selection. At this point, a tabbed dialog should appear. In this new window, choose the *C/C++ compiler* tab. In the edit box, you need to modify the full pathname of the *build_vxworks* subdirectory of Berkeley DB, followed by the full pathname of Berkeley DB. Then, click OK. Note that some versions of Tornado (such as the version for Windows) do not correctly handle relative pathnames in the include paths.

To build and download the Berkeley DB downloadable application for the first time requires several steps:

1. Select the build you are interested in, and right-click. Choose the *Set... as Active Build* selection.

2. Select the build you are interested in, and right-click. Choose the *Dependencies...* selection. Run dependencies over all files in the Berkeley DB project.

3. Select the build you are interested in, and right-click. Choose the *Rebuild All (Berkeley DB.out)* selection.

4. Select the Berkeley DB project name, and right-click. Choose the *Download "Berkeley DB.out"* selection.

Note that the output file listed about will really be listed as *BerkeleyDB20.out* or *BerkeleyDB22.out* depending on which version of Tornado you are running. You need to repeat this procedure for all builds you are interested in building, as well as for all of the utility project builds you want to run.

Building for VxWorks 6.x

Building With Wind River Workbench using the Makefile

In VxWorks6.x, developers use Wind River Workbench for software development. Two makefiles are provided in the db/build_vxworks directory. Use them to build Berkeley DB or Berkeley DB small build, using the build chain provided with Wind River Workbench.

We assume that you have installed all necessary VxWorks6.x software including the Wind River Workbench, and that you know how to use it.

Use the following steps to build Berkeley DB:

1. Setting variables in the Makefile. Open the Makefile and modify the BDB_ROOT variable to the path of your Berkeley DB source tree's root directory. You may need to set other variables when you build on different platforms, such as BUILD_SPEC, DEBUG_MODE, PROJECT_TYPE, build tool flags and BUILD_SPEC specific settings. Please refer to the documentation of the Workbench for a complete list of available values of each variable. You can also find out the list of values by creating a project using the Workbench. Each variable's available values will be listed in the GUI window which assigns values to that variable.

2. Make sure "make" can be found. Basically you need to set the make utility's path to environment variables.

3. Start up the wrenv utility of the Wind River Workbench.

4. In the command console, move to the $(BDB_ROOT)/build_vxworks/ directory, rename the target makefile (Makefile.6x or Makefile.6x.small) to "Makefile", and run "make". The make process will start and create the directory "bdbvxw". It will contain all intermediate object files as well as the final built image "bdbvxw.out".

5. After the "bdbvxw.out" image is built, you can use command tools or the Workbench IDE to download and run it.

6. Test and Verification. There is a dbdemo and test_micro, which you can run to verify whether everything works fine.

VxWorks notes

Berkeley DB currently disallows the DB_TRUNCATE flag to the DB->open() method on VxWorks because the operations this flag represents are not fully supported under VxWorks.

The DB->sync() method is implemented using an ioctl call into the file system driver with the FIOSYNC command. Most, but not all file system drivers support this call. Berkeley DB requires the use of a file system that supports FIOSYNC.

Building and Running the Demo Program

The demo program should be built in a manner very similar to building Berkeley DB. If you want different or additional BSP build specifications you should add them by following the directions indicated in Building for VxWorks 5.4 and 5.5 (page 326).

The demo program can be downloaded and run by calling the entry function **dbdemo** with the pathname of a database to use. The demo program will ask for some input keys. It creates a database and adds those keys into the database, using the reverse of the key as the data value. When complete you can either enter EOF (control-D) or **quit** and the demo program will display all of the key/data items in the database.

Building and Running the Utility Programs

The Berkeley DB utilities can be downloaded and run by calling the function equivalent to the utility's name. The utility functions take a string containing all the supported arguments. The program will then decompose that string into a traditional argc/argv used internally. For

example, to execute db_stat utility on a database within an environment you would execute the following from the windsh prompt. Obviously you would change the pathname and database name to reflect your system.

```
> db_stat "-h /tmp/myenvhome -d mydatabase.db"
```

VxWorks 5.4/5.5: shared memory

The memory on VxWorks is always resident and fully shared among all tasks running on the target. For this reason, the DB_LOCKDOWN flag has no effect and the DB_SYSTEM_MEM flag is implied for any application that does not specify the DB_PRIVATE flag. Note that the DB_SYSTEM_MEM flag requires all applications use a segment ID to ensure the applications do not overwrite each other's database environments: see the DB_ENV->set_shm_key() method for more information.

VxWorks 5.4/5.5: building a small memory footprint library

A default small footprint build is provided. This default provides equivalent to the **--enable-smallbuild** configuration option described in Building a small memory footprint library (page 294). In order to build the small footprint, you should move `db_config.h` aside and copy `db_config_small.h` to `db_config.h`. Then open up the appropriate small workspace file via Tornado and build as usual.

VxWorks FAQ

The third problem is that all tasks will hang on a dosFs semaphore. You should look at **SPR 72063** in the Wind River Systems' Support pages for a more detailed description of this problem.

1. **I get the error "Workspace open failed: This project workspace is an older format.", when trying to open the supplied workspace on Tornado 2.0 under Windows.**

   This error will occur if the files were extracted in a manner that adds a CR/LF to lines in the file. Make sure that you download the Berkeley DB ".zip" version of the Berkeley DB distribution, and, when extracting the Berkeley DB sources, that you use an unzipper program that will not do any conversion.

2. **I sometimes see spurious output errors about temporary directories.**

   These messages are coming from the `stat`(2) function call in VxWorks. Unlike other systems, there may not be a well known temporary directory on the target. Therefore, we highly recommend that all applications use DB_ENV->set_tmp_dir() to specify a temporary directory for the application.

3. **How can I build Berkeley DB without using Tornado?**

   The simplest way to build Berkeley DB without using Tornado is to configure Berkeley DB on a UNIX system, and then use the Makefile and include files generated by that configuration as the starting point for your build. The Makefile and include files are created during configuration, in the current directory, based on your configuration decisions (for example,

debugging vs. non-debugging builds), so you'll need to configure the system for the way you want Berkeley DB to be built.

Additionally, you'll need to account for the slight difference between the set of source files used in a UNIX build and the set used in a VxWorks build. You can use the following command to create a list of the Berkeley DB VxWorks files. The commands assume you are in the build_vxworks directory of the Berkeley DB distribution:

```
% cat > /tmp/files.sed
s/<BEGIN> FILE_//
s/_objects//
^D
% grep FILE_ BerkeleyDB.wpj | grep _objects | sed -f /tmp/files.sed > /tmp/db.files
```

You will then have a template Makefile and include files, and a list of VxWorks-specific source files. You will need to convert this Makefile and list of files into a form that is acceptable to your specific build environment.

4. **Does Berkeley DB use floating point registers?**

   Yes, there are a few places in Berkeley DB where floating point computations are performed. As a result, all applications that call *taskSpawn* should specify the **VX_FP_TASK** option.

5. **Can I run the test suite under VxWorks?**

   The test suite requires the Berkeley DB Tcl library. In turn, this library requires Tcl 8.4 or greater. In order to run the test suite, you would need to port Tcl 8.4 or greater to VxWorks. The Tcl shell included in *windsh* is not adequate for two reasons. First, it is based on Tcl 8.0. Second, it does not include the necessary Tcl components for adding a Tcl extension.

6. **Are all Berkeley DB features available for VxWorks?**

   All Berkeley DB features are available for VxWorks with the exception of the DB_TRUNCATE flag for DB->open(). The underlying mechanism needed for that flag is not available consistently across different file systems for VxWorks.

7. **Are there any constraints using particular filesystem drivers?**

   There are constraints using the dosFs filesystems with Berkeley DB. Namely, you must configure your dosFs filesystem to support long filenames if you are using Berkeley DB logging in your application. The VxWorks' dosFs 1.0 filesystem, by default, uses the old MS-DOS 8.3 file-naming constraints, restricting to 8 character filenames with a 3 character extension. If you have configured with VxWorks' dosFs 2.0 you should be compatible with Windows FAT32 filesystems which supports long filenames.

8. **Are there any dependencies on particular filesystem drivers?**

   There is one dependency on specifics of filesystem drivers in the port of Berkeley DB to VxWorks. Berkeley DB synchronizes data using the FIOSYNC function to ioctl() (another option would have been to use the FIOFLUSH function instead). The FIOSYNC function was chosen because the NFS client driver, nfsDrv, only supports it and doesn't support FIOFLUSH. All

local file systems, as of VxWorks 5.4, support FIOSYNC -- with the exception of rt11fsLib, which only supports FIOFLUSH. To use rt11fsLib, you will need to modify the os/os_fsync.c file to use the FIOFLUSH function; note that rt11fsLib cannot work with NFS clients.

9. **Are there any known filesystem problems?**

   During the course of our internal testing, we came across three problems with the dosFs 2.0 filesystem that warranted patches from Wind River Systems. We strongly recommend you upgrade to dosFs 2.2, **SPR 79795 (x86)** and **SPR 79569 (PPC)** which fixes all of these problems and many more. You should ask Wind River Systems for the patches to these problems if you encounter them and are unable to upgrade to dosFs 2.2.

   The first problem is that files will seem to disappear. You should look at **SPR 31480** in the Wind River Systems' Support pages for a more detailed description of this problem.

   The second problem is a semaphore deadlock within the dosFs filesystem code. Looking at a stack trace via CrossWind, you will see two or more of your application's tasks waiting in semaphore code within dosFs. The patch for this problem is under **SPR 33221** at Wind River Systems. There are several SPR numbers at Wind River Systems that refer to this particular problem.

10. **Are there any filesystems I cannot use?**

    Currently both the Target Server File System (TSFS) and NFS are not able to be used.

    The Target Server File System (TSFS) uses the netDrv driver. This driver does not support any ioctl that allows flushing to the disk, nor does it allow renaming of files via FIORENAME. The NFS file system uses nfsDrv and that driver does not support FIORENAME and cannot be used with Berkeley DB.

11. **What VxWorks primitives are used for mutual exclusion in Berkeley DB?**

    Mutexes inside of Berkeley DB use the basic binary semaphores in VxWorks. The mutexes are created using the FIFO queue type.

12. **What are the implications of VxWorks' mutex implementation using microkernel resources?**

    On VxWorks, the semaphore primitives implementing mutexes consume system resources. Therefore, if an application unexpectedly fails, those resources could leak. Berkeley DB solves this problem by always allocating mutexes in the persistent shared memory regions. Then, if an application fails, running recovery or explicitly removing the database environment by calling the DB_ENV->remove() method will allow Berkeley DB to release those previously held mutex resources. If an application specifies the DB_PRIVATE flag (choosing not to use persistent shared memory), and then fails, mutexes allocated in that private memory may leak their underlying system resources. Therefore, the DB_ENV->open() flag should be used with caution on VxWorks.

Chapter 31. Upgrading from previous versions of Berkeley DB

Library version information

Each release of the Berkeley DB library has a major version number, a minor version number, and a patch number.

The major version number changes only when major portions of the Berkeley DB functionality have been changed. In this case, it may be necessary to significantly modify applications in order to upgrade them to use the new version of the library.

The minor version number changes when Berkeley DB interfaces have changed, and the new release is not entirely backward-compatible with previous releases. To upgrade applications to the new version, they must be recompiled and potentially, minor modifications made (for example, the order of arguments to a function might have changed).

The patch number changes on each release. If only the patch number has changed in a release, applications do not need to be recompiled, and they can be upgraded to the new version by installing the new version of a shared library or by relinking the application to the new version of a static library.

Internal Berkeley DB interfaces may change at any time and during any release, without warning. This means that the library must be entirely recompiled and reinstalled when upgrading to new releases of the library because there is no guarantee that modules from the current version of the library will interact correctly with modules from a previous release.

To retrieve the Berkeley DB version information, applications should use the DB_ENV->version() function. In addition to the previous information, the DB_ENV->version() function returns a string encapsulating the version information, suitable for display to a user.

Upgrading Berkeley DB installations

The following information describes the general process of upgrading Berkeley DB installations. There are four areas to be considered when upgrading Berkeley DB applications and database environments: the application API, the database environment's region files, the underlying database formats, and, in the case of transactional database environments, the log files. The upgrade procedures required depend on whether or not the release is a major or minor release (in which either the major or minor number of the version changed), or a patch release (in which only the patch number in the version changed). Berkeley DB major and minor releases may optionally include changes in all four areas, that is, the application API, region files, database formats, and log files may not be backward-compatible with previous releases.

Each Berkeley DB major or minor release has information in this chapter of the Reference Guide, describing how to upgrade to the new release. The section describes any API changes made in the release. Application maintainers should review the API changes and update their applications as necessary before recompiling with the new release. In addition, each section includes a page specifying whether the log file format or database formats changed in

non-backward-compatible ways as part of the release. Because there are several underlying Berkeley DB database formats, and they do not all necessarily change in the same release, changes to a database format in a release may not affect any particular application. Further, database and log file formats may have changed but be entirely backward-compatible, in which case no upgrade will be necessary.

A Berkeley DB patch release will never modify the API, regions, log files, or database formats in incompatible ways, and so applications need only be relinked (or, in the case of a shared library, pointed at the new version of the shared library) to upgrade to a new release. Note that internal Berkeley DB interfaces may change at any time and in any release (including patch releases) without warning. This means the library must be entirely recompiled and reinstalled when upgrading to new releases of the library because there is no guarantee that modules from one version of the library will interact correctly with modules from another release. We recommend using the same compiler release when building patch releases as was used to build the original release; in the default configuration, the Berkeley DB library shares data structures from underlying shared memory between threads of control, and should the compiler re-order fields or otherwise change those data structures between the two builds, errors may result.

If the release is a patch release, do the following:

1. Shut down the old version of the application.
2. Install the new version of the application by relinking or installing a new version of the Berkeley DB shared library.
3. Restart the application.

Otherwise, if the application **does not** have a Berkeley DB transactional environment, the application may be installed in the field using the following steps:

1. Shut down the old version of the application.
2. Remove any Berkeley DB environment using the DB_ENV->remove() method or an appropriate system utility.
3. Recompile and install the new version of the application.
4. If necessary, upgrade the application's databases. See Database upgrade (page 47) for more information.
5. Restart the application.

Otherwise, if the application has a Berkeley DB transactional environment, but neither the log file nor database formats need upgrading, the application may be installed in the field using the following steps:

1. Shut down the old version of the application.
2. Run recovery on the database environment using the DB_ENV->open() method or the db_recover utility.

3. Remove any Berkeley DB environment using the DB_ENV->remove() method or an appropriate system utility.

4. Recompile and install the new version of the application.

5. Restart the application.

If the application has a Berkeley DB transactional environment, and the log files need upgrading but the databases do not, the application may be installed in the field using the following steps:

1. Shut down the old version of the application.

2. Still using the old version of Berkeley DB, run recovery on the database environment using the DB_ENV->open() utility.

3. If you used the DB_ENV->open() method to run recovery, make sure that the Berkeley DB environment is removed using the DB_ENV->remove() method or an appropriate system utility.

4. Archive the database environment for catastrophic recovery. See Database and log file archival (page 173) for more information.

5. Recompile and install the new version of the application.

6. Force a checkpoint using the DB_ENV->txn_checkpoint() method or the db_checkpoint utility. If you use the db_checkpoint utility, make sure to use the new version of the utility; that is, the version that came with the release of Berkeley DB to which you are upgrading.

7. Remove unnecessary log files from the environment using the `-d` option on the db_archive utility, or from an application which calls the DB_ENV->log_archive() method with the DB_ARCH_REMOVE flag.

   Note that if you are upgrading a replicated application, then you should *not* perform this step until all of the replication sites have been upgraded to the current release level. If you run this site before all your sites are upgraded, then errors can occur in your replication activities because important version information might be lost.

8. Restart the application.

Otherwise, if the application has a Berkeley DB transactional environment and the databases need upgrading, the application may be installed in the field using the following steps:

1. Shut down the old version of the application.

2. Still using the old version of Berkeley DB, run recovery on the database environment using the DB_ENV->open() utility.

3. If you used the DB_ENV->open() method to run recovery, make sure that the Berkeley DB environment is removed using the DB_ENV->remove() method or an appropriate system utility.

4. Archive the database environment for catastrophic recovery. See Database and log file archival (page 173) for more information.

5. Recompile and install the new version of the application.

6. Upgrade the application's databases. See Database upgrade (page 47) for more information.

7. Archive the database for catastrophic recovery again (using different media than before, of course). Note: This archival is not strictly necessary. However, if you have to perform catastrophic recovery after restarting the application, that recovery must be done based on the last archive you have made. If you make this second archive, you can use it as the basis of that catastrophic recovery. If you do not make this second archive, you have to use the archive you made in step 4 as the basis of your recovery, and you have to do a full upgrade on it before you can apply log files created after the upgrade to it.

8. Force a checkpoint using the DB_ENV->txn_checkpoint() method or the db_checkpoint utility. If you use the db_checkpoint utility, make sure to use the new version of the utility; that is, the version that came with the release of Berkeley DB to which you are upgrading.

9. Remove unnecessary log files from the environment using the `-d` option on the db_archive utility, or from an application which calls the DB_ENV->log_archive() method with the DB_ARCH_REMOVE flag.

   Note that if you are upgrading a replicated application, then you should *not* perform this step until all of the replication sites have been upgraded to the current release level. If you run this site before all your sites are upgraded, then errors can occur in your replication activities because important version information might be lost.

10. Restart the application.

Finally, Berkeley DB supports the live upgrade of a replication group, by allowing mixed version operation (replication sites running at the newer software version can inter-operate with older version sites). All client sites must be upgraded first; the master site must be upgraded last. In other words, at all times the master must be running the lowest version of Berkeley DB. To upgrade a replication group, you must:

1. Bring all clients up to date with the master (that is, all clients must be brought up to the most current log record as measured by the master's log sequence number (LSN)).

2. Perform the upgrade procedures described previously on each of the individual database environments that are part of the replication group. Each individual client may be upgraded and restarted to join the replication group.

3. Shut down the master site and upgrade that site last.

During live replication upgrade, while sites are running at different versions, adding new (empty) clients to the replication group is not allowed. Those empty client environments must be added after the entire group is upgraded.

Also, all removal of log files must be suspended throughout this entire procedure, so that there is no chance of a client needing internal initialization.

Alternatively, it may be simpler to discard the contents of all of the client database environments, upgrade the master database environment, and then re-add all of the clients to the replication group using the standard replication procedures for new sites.

Chapter 32. Upgrading Berkeley DB 1.85 or 1.86 applications to Berkeley DB 2.0

Release 2.0: introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 1.85 and 1.86 release interfaces to the Berkeley DB 2.0 release interfaces. They do not describe how to upgrade to the current Berkeley DB release interfaces.

It is not difficult to upgrade Berkeley DB 1.85 applications to use the Berkeley DB version 2 library. The Berkeley DB version 2 library has a Berkeley DB 1.85 compatibility API, which you can use by either recompiling your application's source code or by relinking its object files against the version 2 library. The underlying databases must be converted, however, as the Berkeley DB version 2 library has a different underlying database format.

Release 2.0: system integration

1. It is possible to maintain both the Berkeley DB 1.85 and Berkeley DB version 2 libraries on your system. However, the `db.h` include file that was distributed with Berkeley DB 1.85 is not compatible with the `db.h` file distributed with Berkeley DB version 2, so you will have to install them in different locations. In addition, both the Berkeley DB 1.85 and Berkeley DB version 2 libraries are named `libdb.a`.

   As the Berkeley DB 1.85 library did not have an installation target in the Makefile, there's no way to know exactly where it was installed on the system. In addition, many vendors included it in the C library instead of as a separate library, and so it may actually be part of libc and the `db.h` include file may be installed in `/usr/include`.

   For these reasons, the simplest way to maintain both libraries is to install Berkeley DB version 2 in a completely separate area of your system. The Berkeley DB version 2 installation process allows you to install into a standalone directory hierarchy on your system. See the Building for UNIX/POSIX (page 289) documentation for more information and instructions on how to install the Berkeley DB version 2 library, include files and documentation into specific locations.

2. Alternatively, you can replace Berkeley DB 1.85 on your system with Berkeley DB version 2. In this case, you'll probably want to install Berkeley DB version 2 in the normal place on your system, wherever that may be, and delete the Berkeley DB 1.85 include files, manual pages and libraries.
   To replace 1.85 with version 2, you must either convert your 1.85 applications to use the version 2 API or build the Berkeley DB version 2 library to include Berkeley DB 1.85 interface compatibility code. Whether converting your applications to use the version 2 interface or using the version 1.85 compatibility API, you will need to recompile or relink your 1.85 applications, and you must convert any persistent application databases to the Berkeley DB version 2 database formats.

   If you want to recompile your Berkeley DB 1.85 applications, you will have to change them to include the file `db_185.h` instead of `db.h`. (The `db_185.h` file is automatically installed

during the Berkeley DB version 2 installation process.) You can then recompile the applications, linking them against the Berkeley DB version 2 library.

For more information on compiling the Berkeley DB 1.85 compatibility code into the Berkeley DB version 2 library, see Building for UNIX/POSIX (page 289).

For more information on converting databases from the Berkeley DB 1.85 formats to the Berkeley DB version 2 formats, see the db_dump185 utility and the db_load utility documentation.

3. Finally, although we certainly do not recommend it, it is possible to load both Berkeley DB 1.85 and Berkeley DB version 2 into the same library. Similarly, it is possible to use both Berkeley DB 1.85 and Berkeley DB version 2 within a single application, although it is not possible to use them from within the same file.
   The name space in Berkeley DB version 2 has been changed from that of previous Berkeley DB versions, notably version 1.85, for portability and consistency reasons. The only name collisions in the two libraries are the names used by the historic dbm and hsearch interfaces, and the Berkeley DB 1.85 compatibility interfaces in the Berkeley DB version 2 library.

   If you are loading both Berkeley DB 1.85 and Berkeley DB version 2 into a single library, remove the historic interfaces from one of the two library builds, and configure the Berkeley DB version 2 build to not include the Berkeley DB 1.85 compatibility API, otherwise you could have collisions and undefined behavior. This can be done by editing the library Makefiles and reconfiguring and rebuilding the Berkeley DB version 2 library. Obviously, if you use the historic interfaces, you will get the version in the library from which you did not remove them. Similarly, you will not be able to access Berkeley DB version 2 files using the Berkeley DB 1.85 compatibility interface, since you have removed that from the library as well.

Release 2.0: converting applications

Mapping the Berkeley DB 1.85 functionality into Berkeley DB version 2 is almost always simple. The manual page DB->open() replaces the Berkeley DB 1.85 manual pages `dbopen`(3), `btree`(3), `hash`(3) and `recno`(3). You should be able to convert each 1.85 function call into a Berkeley DB version 2 function call using just the DB->open() documentation.

Some guidelines and things to watch out for:

1. Most access method functions have exactly the same semantics as in Berkeley DB 1.85, although the arguments to the functions have changed in some cases. To get your code to compile, the most common change is to add the transaction ID as an argument (NULL, since Berkeley DB 1.85 did not support transactions.)

2. You must always initialize DBT structures to zero before using them with any Berkeley DB version 2 function. (They do not normally have to be reinitialized each time, only when they are first allocated. Do this by declaring the DBT structure external or static, or by calling the C library routine `bzero`(3) or `memset`(3).)

3. The error returns are completely different in the two versions. In Berkeley DB 1.85, < 0 meant an error, and > 0 meant a minor Berkeley DB exception. In Berkeley DB 2.0, > 0 means

an error (the Berkeley DB version 2 functions return `errno` on error) and < 0 means a Berkeley DB exception. See Error returns to applications (page 229) for more information.

4. The Berkeley DB 1.85 DB->seq function has been replaced by cursors in Berkeley DB version 2. The semantics are approximately the same, but cursors require the creation of an extra object (the DBC object), which is then used to access the database.
   Specifically, the partial key match and range search functionality of the R_CURSOR flag in DB->seq has been replaced by the DB_SET_RANGE flag in DBC->get().

5. In version 2 of the Berkeley DB library, additions or deletions into Recno (fixed and variable-length record) databases no longer automatically logically renumber all records after the add/delete point, by default. The default behavior is that deleting records does not cause subsequent records to be renumbered, and it is an error to attempt to add new records between records already in the database. Applications wanting the historic Recno access method semantics should call the DB->set_flags() method with the DB_RENUMBER flag.

6. Opening a database in Berkeley DB version 2 is a much heavier-weight operation than it was in Berkeley DB 1.85. Therefore, if your historic applications were written to open a database, perform a single operation, and close the database, you may observe performance degradation. In most cases, this is due to the expense of creating the environment upon each open. While we encourage restructuring your application to avoid repeated opens and closes, you can probably recover most of the lost performance by simply using a persistent environment across invocations.

While simply converting Berkeley DB 1.85 function calls to Berkeley DB version 2 function calls will work, we recommend that you eventually reconsider your application's interface to the Berkeley DB database library in light of the additional functionality supplied by Berkeley DB version 2, as it is likely to result in enhanced application performance.

Release 2.0: Upgrade Requirements

You will need to upgrade your on-disk databases, as all access method database formats changed in the Berkeley DB 2.0 release. For information on converting databases from Berkeley DB 1.85 to Berkeley DB 2.0, see the db_dump185 utility and db_load utility documentation. As database environments did not exist prior to the 2.0 release, there is no question of upgrading existing database environments.

Chapter 33. Upgrading Berkeley DB 2.X applications to Berkeley DB 3.0

Release 3.0: introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 2.X release interfaces to the Berkeley DB 3.0 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 3.0: environment open/close/unlink

The hardest part of upgrading your application from a 2.X code base to the 3.0 release is translating the Berkeley DB environment open, close and remove calls.

There were two logical changes in this part of the Berkeley DB interface. First, in Berkeley DB 3.0, there are no longer separate structures that represent each subsystem (for example, DB_LOCKTAB or DB_TXNMGR) and an overall DB_ENV environment structure. Instead there is only the DB_ENV references should be passed around by your application instead of passing around DB_LOCKTAB or DB_TXNMGR references. This is likely to be a simple change for most applications as few applications use the lock_XXX, log_XXX, memp_XXX or txn_XXX interfaces to create Berkeley DB environments.

The second change is that there are no longer separate open, close, and unlink interfaces to the Berkeley DB subsystems. For example, in previous releases, it was possible to open a lock subsystem either using db_appinit or using the lock_open call. In the 3.0 release the XXX_open interfaces to the subsystems have been removed, and subsystems must now be opened using the 3.0 replacement for the db_appinit call.

To upgrade your application, first find each place your application opens, closes and/or removes a Berkeley DB environment. This will be code of the form:

```
db_appinit, db_appexit
lock_open, lock_close, lock_unlink
log_open, log_close, log_unlink
memp_open, memp_close, memp_unlink
txn_open, txn_close, txn_unlink
```

Each of these groups of calls should be replaced with calls to db_env_create(), DB_ENV->open(), DB_ENV->close(), and DB_ENV->remove().

The db_env_create() call and the call to the DB_ENV->open() method replace the db_appinit, lock_open, log_open, memp_open and txn_open calls. The DB_ENV->close() method replaces the db_appexit, lock_close, log_close, memp_close and txn_close calls. The DB_ENV->remove() call replaces the lock_unlink, log_unlink, memp_unlink and txn_unlink calls.

Here's an example creating a Berkeley DB environment using the 2.X interface:

```
/*
 * db_init --
```

```
 * Initialize the environment.
 */
DB_ENV *
db_init(home)
 char *home;
{
 DB_ENV *dbenv;

 if ((dbenv = (DB_ENV *)calloc(sizeof(DB_ENV), 1)) == NULL)
  return (errno);

 if ((errno = db_appinit(home, NULL, dbenv,
     DB_INIT_LOCK | DB_INIT_LOG | DB_INIT_MPOOL | DB_INIT_TXN |
     DB_USE_ENVIRON)) == 0)
  return (dbenv);

 free(dbenv);
 return (NULL);
}
```

In the Berkeley DB 3.0 release, this code would be written as:

```
/*
 * db_init --
 * Initialize the environment.
 */
int
db_init(home, dbenvp)
 char *home;
 DB_ENV **dbenvp;
{
 int ret;
 DB_ENV *dbenv;

 if ((ret = db_env_create(&dbenv, 0)) != 0)
  return (ret);

    if ((ret = dbenv->open(dbenv, home, NULL,
     DB_INIT_LOCK | DB_INIT_LOG | DB_INIT_MPOOL | DB_INIT_TXN |
     DB_USE_ENVIRON, 0)) == 0) {
  *dbenvp = dbenv;
  return (0);
 }

    (void)dbenv->close(dbenv, 0);
 return (ret);
}
```

As you can see, the arguments to db_appinit and to DB_ENV->open() are largely the same. There is some minor re-organization: the mapping is that arguments #1, 2, 3, and 4 to db_appinit

become arguments #2, 3, 1 and 4 to DB_ENV->open(). There is one additional argument to DB_ENV->open(), argument #5. For backward compatibility with the 2.X Berkeley DB releases, simply set that argument to 0.

It is only slightly more complex to translate calls to XXX_open to the DB_ENV->open() method. Here's an example of creating a lock region using the 2.X interface:

```
lock_open(dir, DB_CREATE, 0664, dbenv, &regionp);
```

In the Berkeley DB 3.0 release, this code would be written as:

```
if ((ret = db_env_create(&dbenv, 0)) != 0)
 return (ret);

if ((ret = dbenv->open(dbenv,
    dir, NULL, DB_CREATE | DB_INIT_LOCK, 0664)) == 0) {
 *dbenvp = dbenv;
 return (0);
}
```

Note that in this example, you no longer need the DB_LOCKTAB structure reference that was required in Berkeley DB 2.X releases.

The final issue with upgrading the db_appinit call is the DB_MPOOL_PRIVATE option previously provided for the db_appinit call. If your application is using this flag, it should almost certainly use the new DB_PRIVATE flag to the DB_ENV->open() method. Regardless, you should carefully consider this change before converting to use the DB_PRIVATE flag.

Translating db_appexit or XXX_close calls to DB_ENV->close() is equally simple. Instead of taking a reference to a per-subsystem structure such as DB_LOCKTAB or DB_TXNMGR, all calls take a reference to a DB_ENV structure. The calling sequence is otherwise unchanged. Note that as the application no longer allocates the memory for the DB_ENV structure, application code to discard it after the call to db_appexit() is no longer needed.

Translating XXX_unlink calls to DB_ENV->remove() is slightly more complex. As with DB_ENV->close(), the call takes a reference to a DB_ENV structure instead of a per-subsystem structure. The calling sequence is slightly different, however. Here is an example of removing a lock region using the 2.X interface:

```
DB_ENV *dbenv;

ret = lock_unlink(dir, 1, dbenv);
```

In the Berkeley DB 3.0 release, this code fragment would be written as:

```
DB_ENV *dbenv;

ret = dbenv->remove(dbenv, dir, NULL, DB_FORCE);
```

The additional argument to the DB_ENV->remove() function is a configuration argument similar to that previously taken by db_appinit and now taken by the DB_ENV->open() method. For backward compatibility this new argument should simply be set to NULL. The force argument

to XXX_unlink is now a flag value that is set by bitwise inclusively **OR**'ing it the DB_ENV->remove() flag argument.

Release 3.0: function arguments

In Berkeley DB 3.0, there are no longer separate structures that represent each subsystem (for example, DB_LOCKTAB or DB_TXNMGR), and an overall DB_ENV environment structure. Instead there is only the DB_ENV references should be passed around by your application instead of passing around DB_LOCKTAB or DB_TXNMGR references.

Each of the following functions:

```
lock_detect
lock_get
lock_id
lock_put
lock_stat
lock_vec
```

should have its first argument, a reference to the DB_LOCKTAB structure, replaced with a reference to the enclosing DB_ENV structure. For example, the following line of code from a Berkeley DB 2.X application:

```
DB_LOCKTAB *lt;
DB_LOCK lock;

ret = lock_put(lt, lock);
```

should now be written as follows:

```
DB_ENV *dbenv;
DB_LOCK *lock;

ret = lock_put(dbenv, lock);
```

Similarly, all of the functions:

```
log_archive
log_compare
log_file
log_flush
log_get
log_put
log_register
log_stat
log_unregister
```

should have their DB_LOG argument replaced with a reference to a DB_ENV structure, and the functions:

```
memp_fopen
memp_register
```

```
memp_stat
memp_sync
memp_trickle
```

should have their DB_MPOOL argument replaced with a reference to a DB_ENV structure.

You should remove all references to DB_LOCKTAB, DB_LOG, DB_MPOOL, and DB_TXNMGR structures from your application, they are no longer useful in any way. In fact, a simple way to identify all of the places that need to be upgraded is to remove all such structures and variables they declare, and then compile. You will see a warning message from your compiler in each case that needs to be upgraded.

Release 3.0: the DB_ENV structure

The DB_ENV structure is now opaque for applications in the Berkeley DB 3.0 release. Accesses to any fields within that structure by the application should be replaced with method calls. The following example illustrates this using the historic errpfx structure field. In the Berkeley DB 2.X releases, applications set error prefixes using code similar to the following:

```
DB_ENV *dbenv;

dbenv->errpfx = "my prefix";
```

in the Berkeley DB 3.X releases, this should be done using the DB_ENV->set_errpfx() method, as follows:

```
DB_ENV *dbenv;

dbenv->set_errpfx(dbenv, "my prefix");
```

The following table lists the DB_ENV fields previously used by applications and the methods that should now be used to set them.

DB_ENV field	Berkeley DB 3.X method
db_errcall	DB_ENV->set_errcall()
db_errfile	DB_ENV->set_errfile()
db_errpfx	DB_ENV->set_errpfx()
db_lorder	This field was removed from the DB_ENV structure in the Berkeley DB 3.0 release as no application should have ever used it. Any code using it should be evaluated for potential bugs.
db_paniccall	DB_ENV->set_paniccall
db_verbose	DB_ENV->set_verbose() Note: the db_verbose field was a simple boolean toggle, the DB_ENV->set_verbose() method takes arguments that specify exactly which verbose messages are desired.

DB_ENV field	Berkeley DB 3.X method
lg_max	DB_ENV->set_lg_max()
lk_conflicts	DB_ENV->set_lk_conflicts()
lk_detect	DB_ENV->set_lk_detect()
lk_max	dbenv->set_lk_max
lk_modes	DB_ENV->set_lk_conflicts()
mp_mmapsize	DB_ENV->set_mp_mmapsize()
mp_size	DB_ENV->set_cachesize() Note: the DB_ENV->set_cachesize() function takes additional arguments. Setting both the second argument (the number of GB in the pool) and the last argument (the number of memory pools to create) to 0 will result in behavior that is backward-compatible with previous Berkeley DB releases.
tx_info	This field was used by applications as an argument to the transaction subsystem functions. As those functions take references to a DB_ENV structure as arguments in the Berkeley DB 3.0 release, it should no longer be used by any application.
tx_max	DB_ENV->set_tx_max()
tx_recover	dbenv->set_tx_recover

Release 3.0: database open/close

Database opens were changed in the Berkeley DB 3.0 release in a similar way to environment opens.

To upgrade your application, first find each place your application opens a database, that is, calls the db_open function. Each of these calls should be replaced with calls to db_create() and DB->open().

Here's an example creating a Berkeley DB database using the 2.X interface:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = db_open(DATABASE,
    DB_BTREE, DB_CREATE, 0664, dbenv, NULL, &dbp)) != 0)
 return (ret);
```

In the Berkeley DB 3.0 release, this code would be written as:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = db_create(&dbp, dbenv, 0)) != 0)
 return (ret);

if ((ret = dbp->open(dbp,
    DATABASE, NULL, DB_BTREE, DB_CREATE, 0664)) != 0) {
    (void)dbp->close(dbp, 0);
 return (ret);
}
```

As you can see, the arguments to db_open and to DB->open() are largely the same. There is some re-organization, and note that the enclosing DB_ENV structure is specified when the DB object is created using the db_create() function. There is one additional argument to DB->open(), argument #3. For backward compatibility with the 2.X Berkeley DB releases, simply set that argument to NULL.

There are two additional issues with the db_open call.

First, it was possible in the 2.X releases for an application to provide an environment that did not contain a shared memory buffer pool as the database environment, and Berkeley DB would create a private one automatically. This functionality is no longer available, applications must specify the DB_INIT_MPOOL flag if databases are going to be opened in the environment.

The final issue with upgrading the db_open call is that the DB_INFO structure is no longer used, having been replaced by individual methods on the DB handle. That change is discussed in detail later in this chapter.

Release 3.0: db_xa_open

The following change applies only to applications using Berkeley DB as an XA Resource Manager. If your application is not using Berkeley DB in this way, you can ignore this change.

The db_xa_open function has been replaced with the DB_XA_CREATE flag to the db_create() function. All calls to db_xa_open should be replaced with calls to db_create() with the DB_XA_CREATE flag set, followed by a call to the DB->open() function.

A similar change has been made for the C++ API, where the DB_XA_CREATE flag should be specified to the Db constructor. All calls to the Db::xa_open method should be replaced with the DB_XA_CREATE flag to the Db constructor, followed by a call to the DB::open method.

Release 3.0: the DB structure

The DB structure is now opaque for applications in the Berkeley DB 3.0 release. Accesses to any fields within that structure by the application should be replaced with method calls. The following example illustrates this using the historic type structure field. In the Berkeley DB 2.X

releases, applications could find the type of an underlying database using code similar to the following:

```
DB *db;
DB_TYPE type;

type = db->type;
```

in the Berkeley DB 3.X releases, this should be done using the DB->get_type() method, as follows:

```
DB *db;
DB_TYPE type;

type = db->get_type(db);
```

The following table lists the DB fields previously used by applications and the methods that should now be used to get or set them.

DB field	Berkeley DB 3.X method
byteswapped	DB->get_byteswapped()
db_errcall	DB->set_errcall()
db_errfile	DB->set_errfile()
db_errpfx	DB->set_errpfx()
db_paniccall	DB->set_paniccall
type	DB->get_type()

Release 3.0: the DBINFO structure

The DB_INFO structure has been removed from the Berkeley DB 3.0 release. Accesses to any fields within that structure by the application should be replaced with method calls on the DB handle. The following example illustrates this using the historic db_cachesize structure field. In the Berkeley DB 2.X releases, applications could set the size of an underlying database cache using code similar to the following:

```
DB_INFO dbinfo;

memset(dbinfo, 0, sizeof(dbinfo));
dbinfo.db_cachesize = 1024 * 1024;
```

in the Berkeley DB 3.X releases, this should be done using the DB->set_cachesize() method, as follows:

```
DB *db;
int ret;

ret = db->set_cachesize(db, 0, 1024 * 1024, 0);
```

The DB_INFO structure is no longer used in any way by the Berkeley DB 3.0 release, and should be removed from the application.

The following table lists the DB_INFO fields previously used by applications and the methods that should now be used to set them. Because these calls provide configuration for the database open, they must precede the call to DB->open(). Calling them after the call to DB->open() will return an error.

DB_INFO field	Berkeley DB 3.X method
bt_compare	DB->set_bt_compare()
bt_minkey	DB->set_bt_minkey()
bt_prefix	DB->set_bt_prefix()
db_cachesize	DB->set_cachesize() Note: the DB->set_cachesize() function takes additional arguments. Setting both the second argument (the number of GB in the pool) and the last argument (the number of memory pools to create) to 0 will result in behavior that is backward-compatible with previous Berkeley DB releases.
db_lorder	DB->set_lorder()
db_malloc	DB->set_malloc
db_pagesize	DB->set_pagesize()
dup_compare	DB->set_dup_compare()
flags	DB->set_flags() Note: the DB_DELIMITER, DB_FIXEDLEN and DB_PAD flags no longer need to be set as there are specific methods off the DB handle that set the file delimiter, the length of fixed-length records and the fixed-length record pad character. They should simply be discarded from the application.
h_ffactor	DB->set_h_ffactor()
h_hash	DB->set_h_hash()
h_nelem	DB->set_h_nelem()
re_delim	DB->set_re_delim()
re_len	DB->set_re_len()
re_pad	DB->set_re_pad()
re_source	DB->set_re_source()

Release 3.0: DB->join

Historically, the last two arguments to the DB->join() method were a flags value followed by a reference to a memory location to store the returned cursor object. In the Berkeley DB 3.0 release, the order of those two arguments has been swapped for consistency with other Berkeley DB interfaces.

The application should be searched for any occurrences of DB->join(). For each of these, the order of the last two arguments should be swapped.

Release 3.0: DB->stat

The **bt_flags** field returned from the DB->stat() method for Btree and Recno databases has been removed, and this information is no longer available.

Release 3.0: DB->sync and DB->close

In previous Berkeley DB releases, the DB->close() and DB->sync() methods discarded any return of DB_INCOMPLETE from the underlying buffer pool interfaces, and returned success to its caller. (The DB_INCOMPLETE error will be returned if the buffer pool functions are unable to flush all of the database's dirty blocks from the pool. This often happens if another thread is reading or writing the database's pages in the pool.)

In the 3.X release, DB->sync() and DB->close() will return DB_INCOMPLETE to the application. The best solution is to not call DB->sync() with the DB_NOSYNC flag to the DB->close() method when multiple threads are expected to be accessing the database. Alternatively, the caller can ignore any error return of DB_INCOMPLETE.

Release 3.0: lock_put

An argument change has been made in the lock_put function.

The application should be searched for any occurrences of lock_put. For each one, instead of passing a DB_LOCK variable as the last argument to the function, the address of the DB_LOCK variable should be passed.

Release 3.0: lock_detect

An additional argument has been added to the lock_detect function.

The application should be searched for any occurrences of lock_detect. For each one, a NULL argument should be appended to the current arguments.

Release 3.0: lock_stat

The **st_magic**, **st_version**, **st_numobjs** and **st_refcnt** fields returned from the lock_stat function have been removed, and this information is no longer available.

Release 3.0: log_register

An argument has been removed from the log_register function. The application should be searched for any occurrences of log_register. In each of these, the DBTYPE argument (it is the fourth argument) should be removed.

Release 3.0: log_stat

The **st_refcnt** field returned from the log_stat function has been removed, and this information is no longer available.

Release 3.0: memp_stat

The **st_refcnt** field returned from the memp_stat function has been removed, and this information is no longer available.

The **st_cachesize** field returned from the memp_stat function has been replaced with two new fields, **st_gbytes** and **st_bytes**.

Release 3.0: txn_begin

An additional argument has been added to the txn_begin function.

The application should be searched for any occurrences of txn_begin. For each one, an argument of 0 should be appended to the current arguments.

Release 3.0: txn_commit

An additional argument has been added to the txn_commit function.

The application should be searched for any occurrences of txn_commit. For each one, an argument of 0 should be appended to the current arguments.

Release 3.0: txn_stat

The **st_refcnt** field returned from the txn_stat function has been removed, and this information is no longer available.

Release 3.0: DB_RMW

The following change applies only to applications using the Berkeley DB Concurrent Data Store product. If your application is not using that product, you can ignore this change.

Historically, the DB->cursor() method took the DB_RMW flag to indicate that the created cursor would be used for write operations on the database. This flag has been renamed to the `DB_WRITECURSOR` flag.

The application should be searched for any occurrences of DB_RMW. For each of these, any that are arguments to the DB->cursor() function should be changed to pass in the DB_WRITECURSOR flag instead.

Release 3.0: DB_LOCK_NOTHELD

Historically, the Berkeley DB lock_put and lock_vec interfaces could return the DB_LOCK_NOTHELD error to indicate that a lock could not be released as it was held by another locker. This error can no longer be returned under any circumstances. The application should be searched for any occurrences of DB_LOCK_NOTHELD. For each of these, the test and any error processing should be removed.

Release 3.0: EAGAIN

Historically, the Berkeley DB interfaces have returned the POSIX error value EAGAIN to indicate a deadlock. This has been removed from the Berkeley DB 3.0 release in order to make it possible for applications to distinguish between EAGAIN errors returned by the system and returns from Berkeley DB indicating deadlock.

The application should be searched for any occurrences of EAGAIN. For each of these, any that are checking for a deadlock return from Berkeley DB should be changed to check for the DB_LOCK_DEADLOCK return value.

If, for any reason, this is a difficult change for the application to make, the `include/db.src` distribution file should be modified to translate all returns of DB_LOCK_DEADLOCK to EAGAIN. Search for the string EAGAIN in that file, there is a comment that describes how to make the change.

Release 3.0: EACCES

There was an error in previous releases of the Berkeley DB documentation that said that the lock_put and lock_vec interfaces could return EACCES as an error to indicate that a lock could not be released because it was held by another locker. The application should be searched for any occurrences of EACCES. For each of these, any that are checking for an error return from lock_put or lock_vec should have the test and any error handling removed.

Release 3.0: db_jump_set

The db_jump_set interface has been removed from the Berkeley DB 3.0 release, replaced by method calls on the DB_ENV handle.

The following table lists the db_jump_set arguments previously used by applications and the methods that should now be used instead.

db_jump_set argument	Berkeley DB 3.X method
DB_FUNC_CLOSE	db_env_set_func_close
DB_FUNC_DIRFREE	db_env_set_func_dirfree

db_jump_set argument	Berkeley DB 3.X method
DB_FUNC_DIRLIST	db_env_set_func_dirlist
DB_FUNC_EXISTS	db_env_set_func_exists
DB_FUNC_FREE	db_env_set_func_free
DB_FUNC_FSYNC	db_env_set_func_fsync
DB_FUNC_IOINFO	db_env_set_func_ioinfo
DB_FUNC_MALLOC	db_env_set_func_malloc
DB_FUNC_MAP	dbenv_set_func_map
DB_FUNC_OPEN	db_env_set_func_open
DB_FUNC_READ	db_env_set_func_read
DB_FUNC_REALLOC	db_env_set_func_realloc
DB_FUNC_RUNLINK	The DB_FUNC_RUNLINK functionality has been removed from the Berkeley DB 3.0 release, and should be removed from the application.
DB_FUNC_SEEK	db_env_set_func_seek
DB_FUNC_SLEEP	db_env_set_func_sleep
DB_FUNC_UNLINK	db_env_set_func_unlink
DB_FUNC_UNMAP	dbenv_set_func_unmap
DB_FUNC_WRITE	db_env_set_func_write
DB_FUNC_YIELD	db_env_set_func_yield

Release 3.0: db_value_set

The db_value_set function has been removed from the Berkeley DB 3.0 release, replaced by method calls on the DB_ENV handle.

The following table lists the db_value_set arguments previously used by applications and the function that should now be used instead.

db_value_set argument	Berkeley DB 3.X method
DB_MUTEX_LOCKS	dbenv_set_mutexlocks
DB_REGION_ANON	The DB_REGION_ANON functionality has been replaced by the DB_SYSTEM_MEM and DB_PRIVATE flags to the DB_ENV->open() function. A direct translation is not available, please review the DB_ENV->open() manual page for more information.
DB_REGION_INIT	dbenv_set_region_init
DB_REGION_NAME	The DB_REGION_NAME functionality has been replaced by the DB_SYSTEM_MEM and

db_value_set argument	Berkeley DB 3.X method
	DB_PRIVATE flags to the DB_ENV->open() function. A direct translation is not available, please review the DB_ENV->open() manual page for more information.
DB_TSL_SPINS	dbenv_set_tas_spins

Release 3.0: the DbEnv class for C++ and Java

The DbEnv::appinit() method and two constructors for the DbEnv class are gone. There is now a single way to create and initialize the environment. The way to create an environment is to use the new DbEnv constructor with one argument. After this call, the DbEnv can be configured with various set_XXX methods. Finally, a call to DbEnv::open is made to initialize the environment.

Here's a C++ example creating a Berkeley DB environment using the 2.X interface

```
int dberr;
DbEnv *dbenv = new DbEnv();

dbenv->set_error_stream(&cerr);
dbenv->set_errpfx("myprog");

if ((dberr = dbenv->appinit("/database/home",
 NULL, DB_CREATE | DB_INIT_LOCK | DB_INIT_MPOOL)) != 0) {
    cerr << "failure: " << strerror(dberr);
    exit (1);
}
```

In the Berkeley DB 3.0 release, this code would be written as:

```
int dberr;
DbEnv *dbenv = new DbEnv(0);

dbenv->set_error_stream(&cerr);
dbenv->set_errpfx("myprog");

if ((dberr = dbenv->open("/database/home",
 NULL, DB_CREATE | DB_INIT_LOCK | DB_INIT_MPOOL, 0)) != 0) {
    cerr << "failure: " << dbenv->strerror(dberr);
    exit (1);
}
```

Here's a Java example creating a Berkeley DB environment using the 2.X interface:

```
int dberr;
DbEnv dbenv = new DbEnv();

dbenv.set_error_stream(System.err);
```

```
dbenv.set_errpfx("myprog");

dbenv.appinit("/database/home",
    null, Db.DB_CREATE | Db.DB_INIT_LOCK | Db.DB_INIT_MPOOL);
```

In the Berkeley DB 3.0 release, this code would be written as:

```
int dberr;
DbEnv dbenv = new DbEnv(0);

dbenv.set_error_stream(System.err);
dbenv.set_errpfx("myprog");

dbenv.open("/database/home",
    null, Db.DB_CREATE | Db.DB_INIT_LOCK | Db.DB_INIT_MPOOL, 0);
```

In the Berkeley DB 2.X release, DbEnv had accessors to obtain "managers" of type DbTxnMgr, DbMpool, DbLog, DbTxnMgr. If you used any of these managers, all their methods are now found directly in the DbEnv class.

Release 3.0: the Db class for C++ and Java

The static Db::open method and the DbInfo class have been removed in the Berkeley DB 3.0 release. The way to open a database file is to use the new Db constructor with two arguments, followed by set_XXX methods to configure the Db object, and finally a call to the new (nonstatic) Db::open(). In comparing the Berkeley DB 3.0 release open method with the 2.X static open method, the second argument is new. It is a database name, which can be null. The DbEnv argument has been removed, as the environment is now specified in the constructor. The open method no longer returns a Db, since it operates on one.

Here's a C++ example opening a Berkeley DB database using the 2.X interface:

```
// Note: by default, errors are thrown as exceptions
Db *table;
Db::open("lookup.db", DB_BTREE, DB_CREATE, 0644, dbenv, 0, &table);
```

In the Berkeley DB 3.0 release, this code would be written as:

```
// Note: by default, errors are thrown as exceptions
Db *table = new Db(dbenv, 0);
table->open("lookup.db", NULL, DB_BTREE, DB_CREATE, 0644);
```

Here's a Java example opening a Berkeley DB database using the 2.X interface:

```
// Note: errors are thrown as exceptions
Db table = Db.open("lookup.db", Db.DB_BTREE, Db.DB_CREATE, 0644, dbenv, 0);
```

In the Berkeley DB 3.0 release, this code would be written as:

```
// Note: errors are thrown as exceptions
Db table = new Db(dbenv, 0);
table.open("lookup.db", null, Db.DB_BTREE, Db.DB_CREATE, 0644);
```

Note that if the dbenv argument is null, the database will not exist within an environment.

Release 3.0: additional C++ changes

The Db::set_error_model method is gone. The way to change the C++ API to return errors rather than throw exceptions is via a flag on the DbEnv or Db constructor. For example:

```
int dberr;
DbEnv *dbenv = new DbEnv(DB_CXX_NO_EXCEPTIONS);
```

creates an environment that will never throw exceptions, and method returns should be checked instead.

There are a number of smaller changes to the API that bring the C, C++ and Java APIs much closer in terms of functionality and usage. Please refer to the pages for upgrading C applications for further details.

Release 3.0: additional Java changes

There are several additional types of exceptions thrown in the Berkeley DB 3.0 Java API.

DbMemoryException and DbDeadlockException can be caught independently of DbException if you want to do special handling for these kinds of errors. Since they are subclassed from DbException, a try block that catches DbException will catch these also, so code is not required to change. The catch clause for these new exceptions should appear before the catch clause for DbException.

You will need to add a catch clause for java.io.FileNotFoundException, since that can be thrown by Db.open and DbEnv.open.

There are a number of smaller changes to the API that bring the C, C++ and Java APIs much closer in terms of functionality and usage. Please refer to the pages for upgrading C applications for further details.

Release 3.0: Upgrade Requirements

Log file formats and the Btree, Recno and Hash Access Method database formats changed in the Berkeley DB 3.0 release. (The on-disk Btree/Recno format changed from version 6 to version 7. The on-disk Hash format changed from version 5 to version 6.) Until the underlying databases are upgraded, the DB->open() method will return a `DB_OLD_VERSION` error.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Chapter 34. Upgrading Berkeley DB 3.0 applications to Berkeley DB 3.1

Release 3.1: introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 3.0 release interfaces to the Berkeley DB 3.1 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 3.1: DB_ENV->open, DB_ENV->remove

In the Berkeley DB 3.1 release, the **config** argument to the DB_ENV->open() and DB_ENV->remove() methods has been removed, replaced by additional methods on the DB_ENV handle. If your application calls DB_ENV->open() or DB_ENV->remove() with a NULL **config** argument, find those functions and remove the config argument from the call. If your application has non-NULL **config** argument, the strings values in that argument are replaced with calls to DB_ENV methods as follows:

Previous config string	Berkeley DB 3.1 version method
DB_DATA_DIR	DB_ENV->set_data_dir()
DB_LOG_DIR	DB_ENV->set_lg_dir()
DB_TMP_DIR	DB_ENV->set_tmp_dir()

Release 3.1: DB_ENV->set_tx_recover

The redo parameter of the function passed to DB_ENV->set_tx_recover used to be an integer set to any one of a number of #defined values. In the 3.1 release of Berkeley DB, the redo parameter has been replaced by the op parameter which is an enumerated type of type db_recops.

If your application calls DB_ENV->set_tx_recover, then find the function referred to by the call. Replace the flag values in that function as follows:

Previous flag	Berkeley DB 3.1 version flag
TXN_BACKWARD_ROLL	DB_TXN_BACKWARD_ROLL
TXN_FORWARD_ROLL	DB_TXN_FORWARD_ROLL
TXN_OPENFILES	DB_TXN_OPENFILES
TXN_REDO	DB_TXN_FORWARD_ROLL
TXN_UNDO	DB_TXN_ABORT

Release 3.1: DB_ENV->set_feedback, DB->set_feedback

Starting with the 3.1 release of Berkeley DB, the DB_ENV->set_feedback() and DB->set_feedback() methods may return an error value, that is, they are no longer declared as returning no value, instead they return an int or throw an exception as appropriate when an error occurs.

If your application calls these functions, you may want to check for a possible error on return.

Release 3.1: DB_ENV->set_paniccall, DB->set_paniccall

Starting with the 3.1 release of Berkeley DB, the DB_ENV->set_paniccall and DB->set_paniccall methods may return an error value, that is, they are no longer declared as returning no value, instead they return an int or throw an exception as appropriate when an error occurs.

If your application calls these functions, you may want to check for a possible error on return.

Release 3.1: DB->put

For the Queue and Recno access methods, when the DB_APPEND flag is specified to the DB->put() method, the allocated record number is returned to the application in the **key** DBT argument. In previous releases of Berkeley DB, this DBT structure did not follow the usual DBT conventions. For example, it was not possible to cause Berkeley DB to allocate space for the returned record number. Rather, it was always assumed that the **data** field of the **key** structure referred to memory that could be used as storage for a db_recno_t type.

As of the Berkeley DB 3.1.0 release, the **key** structure behaves as described in the DBT C++/Java class or C structure documentation.

Applications which are using the DB_APPEND flag for Queue and Recno access method databases will require a change to upgrade to the Berkeley DB 3.1 releases. The simplest change is likely to be to add the DB_DBT_USERMEM flag to the **key** structure. For example, code that appears as follows:

```
DBT key;
db_recno_t recno;

memset(&key, 0, sizeof(DBT));
key.data = &recno;
key.size = sizeof(recno);
DB->put(DB, NULL, &key, &data, DB_APPEND);
printf("new record number is %lu\n", (u_long)recno);
```

would be changed to:

```
DBT key;
db_recno_t recno;

memset(&key, 0, sizeof(DBT));
key.data = &recno;
```

```
key.ulen = sizeof(recno);
key.flags = DB_DBT_USERMEM;
DB->put(DB, NULL, &key, &data, DB_APPEND);
printf("new record number is %lu\n", (u_long)recno);
```

Note that the **ulen** field is now set as well as the flag value. An alternative change would be:

```
DBT key;
db_recno_t recno;

memset(&key, 0, sizeof(DBT));
DB->put(DB, NULL, &key, &data, DB_APPEND);
recno = *(db_recno_t *)key->data;
printf("new record number is %lu\n", (u_long)recno);
```

Release 3.1: identical duplicate data items

In previous releases of Berkeley DB, it was not an error to store identical duplicate data items, or, for those that just like the way it sounds, duplicate duplicates. However, there were implementation bugs where storing duplicate duplicates could cause database corruption.

In this release, applications may store identical duplicate data items as long as the data items are unsorted. 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. See Duplicate data items (page 22) for more information.

Release 3.1: DB->stat

For Btree database statistics, the DB->stat() method field **bt_nrecs** has been removed, replaced by two fields: **bt_nkeys** and **bt_ndata**. The **bt_nkeys** field returns a count of the unique keys in the database. The **bt_ndata** field returns a count of the key/data pairs in the database. Neither exactly matches the previous value of the **bt_nrecs** field, which returned a count of keys in the database, but, in the case of Btree databases, could overcount as it sometimes counted duplicate data items as unique keys. The application should be searched for any uses of the **bt_nrecs** field and the field should be changed to be either **bt_nkeys** or **bt_ndata**, whichever is more appropriate.

For Hash database statistics, the DB->stat() method field **hash_nrecs** has been removed, replaced by two fields: **hash_nkeys** and **hash_ndata**. The **hash_nkeys** field returns a count of the unique keys in the database. The **hash_ndata** field returns a count of the key/data pairs in the database. The new **hash_nkeys** field exactly matches the previous value of the **hash_nrecs** field. The application should be searched for any uses of the **hash_nrecs** field, and the field should be changed to be **hash_nkeys**.

For Queue database statistics, the DB->stat() method field **qs_nrecs** has been removed, replaced by two fields: **qs_nkeys** and **qs_ndata**. The **qs_nkeys** field returns a count of the unique keys in the database. The **qs_ndata** field returns a count of the key/data pairs in the database. The new **qs_nkeys** field exactly matches the previous value of the **qs_nrecs** field. The application should be searched for any uses of the **qs_nrecs** field, and the field should be changed to be **qs_nkeys**.

Release 3.1: DB_SYSTEM_MEM

Using the DB_SYSTEM_MEM option on UNIX systems now requires the specification of a base system memory segment ID, using the DB_ENV->set_shm_key() method. Any valid segment ID may be specified, for example, one returned by the UNIX `ftok`(3) function.

Release 3.1: log_register

The arguments to the log_register and log_unregister interfaces have changed. Instead of returning (and passing in) a logging file ID, a reference to the DB structure being registered (or unregistered) is passed. The application should be searched for any occurrences of log_register and log_unregister. For each one, change the arguments to be a reference to the DB structure being registered or unregistered.

Release 3.1: memp_register

An additional argument has been added to the **pgin** and **pgout** functions provided to the memp_register function. The application should be searched for any occurrences of memp_register. For each one, if **pgin** or **pgout** functions are specified, the **pgin** and **pgout** functions should be modified to take an initial argument of a **DB_ENV ***. This argument is intended to support better error reporting for applications, and may be entirely ignored by the **pgin** and **pgout** functions themselves.

Release 3.1: txn_checkpoint

An additional argument has been added to the txn_checkpoint function.

The application should be searched for any occurrences of txn_checkpoint. For each one, an argument of 0 should be appended to the current arguments.

Release 3.1: environment configuration

A set of DB_ENV configuration methods which were not environment specific, but which instead affected the entire application space, have been removed from the DB_ENV object and replaced by static functions. The following table lists the DB_ENV methods previously available to applications and the static functions that should now be used instead.

DB_ENV method	Berkeley DB 3.1 function
DB_ENV->set_func_close	db_env_set_func_close
DB_ENV->set_func_dirfree	db_env_set_func_dirfree
DB_ENV->set_func_dirlist	db_env_set_func_dirlist
DB_ENV->set_func_exists	db_env_set_func_exists
DB_ENV->set_func_free	db_env_set_func_free
DB_ENV->set_func_fsync	db_env_set_func_fsync
DB_ENV->set_func_ioinfo	db_env_set_func_ioinfo

DB_ENV method	Berkeley DB 3.1 function
DB_ENV->set_func_malloc	db_env_set_func_malloc
DB_ENV->set_func_map	dbenv_set_func_map
DB_ENV->set_func_open	db_env_set_func_open
DB_ENV->set_func_read	db_env_set_func_read
DB_ENV->set_func_realloc	db_env_set_func_realloc
DB_ENV->set_func_rename	db_env_set_func_rename
DB_ENV->set_func_seek	db_env_set_func_seek
DB_ENV->set_func_sleep	db_env_set_func_sleep
DB_ENV->set_func_unlink	db_env_set_func_unlink
DB_ENV->set_func_unmap	dbenv_set_func_unmap
DB_ENV->set_func_write	db_env_set_func_write
DB_ENV->set_func_yield	db_env_set_func_yield
DB_ENV->set_pageyield	dbenv_set_pageyield
DB_ENV->set_region_init	dbenv_set_region_init
DB_ENV->set_mutexlocks	dbenv_set_mutexlocks
DB_ENV->set_tas_spins	dbenv_set_tas_spins

Release 3.1: Tcl API

The Berkeley DB Tcl API has been modified so that the **-mpool** option to the **berkdb env** command is now the default behavior. The Tcl API has also been modified so that the **-txn** option to the **berkdb env** command implies the **-lock** and **-log** options. Tcl scripts should be updated to remove the **-mpool**, **-lock** and **-log** options.

The Berkeley DB Tcl API has been modified to follow the Tcl standard rules for integer conversion, for example, if the first two characters of a record number are "0x", the record number is expected to be in hexadecimal form.

Release 3.1: DB_TMP_DIR

This change only affects Win/32 applications that create in-memory databases.

On Win/32 platforms an additional test has been added when searching for the appropriate directory in which to create the temporary files that are used to back in-memory databases. Berkeley DB now uses any return value from the GetTempPath interface as the temporary file directory name before resorting to the static list of compiled-in pathnames.

If the system registry does not return the same directory as Berkeley DB has been using previously, this change could cause temporary backing files to move to a new directory when applications are upgraded to the 3.1 release. In extreme cases, this could create (or fix) security problems if the file protection modes for the system registry directory are different from those on the directory previously used by Berkeley DB.

Release 3.1: log file pre-allocation

This change only affects Win/32 applications.

On Win/32 platforms Berkeley DB no longer pre-allocates log files. The problem was a noticeable performance spike as each log file was created. To turn this feature back on, search for the flag DB_OSO_LOG in the source file `log/log_put.c` and make the change described there, or contact us for assistance.

Release 3.1: Upgrade Requirements

Log file formats and the Btree, Queue, Recno and Hash Access Method database formats changed in the Berkeley DB 3.1 release. (The on-disk Btree/Recno format changed from version 7 to version 8. The on-disk Hash format changed from version 6 to version 7. The on-disk Queue format changed from version 1 to version 2.) Until the underlying databases are upgraded, the DB->open() method will return a `DB_OLD_VERSION` error.

An additional flag, DB_DUPSORT, has been added to the DB->upgrade() method for this upgrade. Please review the DB->upgrade() documentation for further information.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Chapter 35. Upgrading Berkeley DB 3.1 applications to Berkeley DB 3.2

Release 3.2: introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 3.1 release interfaces to the Berkeley DB 3.2 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 3.2: DB_ENV->set_flags

A new method has been added to the Berkeley DB environment handle, DB_ENV->set_flags(). This method currently takes three flags: DB_CDB_ALLDB, DB_NOMMAP, and DB_TXN_NOSYNC. The first of these flags, DB_CDB_ALLDB, provides new functionality, allowing Berkeley DB Concurrent Data Store applications to do locking across multiple databases.

The other two flags, DB_NOMMAP and DB_TXN_NOSYNC, were specified to the DB_ENV->open() method in previous releases. In the 3.2 release, they have been moved to the DB_ENV->set_flags() method because this allows the database environment's value to be toggled during the life of the application as well as because it is a more appropriate place for them. Applications specifying either the DB_NOMMAP or DB_TXN_NOSYNC flags to the DB_ENV->open() method should replace those flags with calls to the DB_ENV->set_flags() method.

Release 3.2: DB callback functions, app_private field

In the Berkeley DB 3.2 release, four application callback functions (the callback functions set by DB->set_bt_compare(), DB->set_bt_prefix(), DB->set_dup_compare() and DB->set_h_hash()) were modified to take a reference to a DB object as their first argument. This change allows the Berkeley DB Java API to reasonably support these interfaces. There is currently no need for the callback functions to do anything with this additional argument.

C and C++ applications that specify their own Btree key comparison, Btree prefix comparison, duplicate data item comparison or Hash functions should modify these functions to take a reference to a DB structure as their first argument. No further change is required.

The app_private field of the DBT structure (accessible only from the Berkeley DB C API) has been removed in the 3.2 release. It was replaced with app_private fields in the DB_ENV handles. Applications using this field will have to convert to using one of the replacement fields.

Release 3.2: Logically renumbering records

In the Berkeley DB 3.2 release, cursor adjustment semantics changed for Recno databases with mutable record numbers. Before the 3.2 release, cursors were adjusted to point to the previous or next record at the time the record to which the cursor referred was deleted. This could lead to unexpected behaviors. For example, two cursors referring to sequential records that were both deleted would lose their relationship to each other and would refer to the same position in the database instead of their original sequential relationship. There were also command

sequences that would have unexpected results. For example, DB_AFTER and DB_BEFORE cursor put operations, using a cursor previously used to delete an item, would perform the put relative to the cursor's adjusted position and not its original position.

In the Berkeley DB 3.2 release, cursors maintain their position in the tree regardless of deletion operations using the cursor. Applications that perform database operations, using cursors previously used to delete entries in Recno databases with mutable record numbers, should be evaluated to ensure that the new semantics do not cause application failure.

Release 3.2: DB_INCOMPLETE

There are a number of functions that flush pages from the Berkeley DB shared memory buffer pool to disk. Most of those functions can potentially fail because a page that needs to be flushed is not currently available. However, this is not a hard failure and is rarely cause for concern. In the Berkeley DB 3.2 release, the C++ API (if that API is configured to throw exceptions) and the Java API have been changed so that this failure does not throw an exception, but rather returns a non-zero error code of `DB_INCOMPLETE`.

The following C++ methods will return `DB_INCOMPLETE` rather than throw an exception: Db::close, Db::sync, DbEnv::memp_sync, DbEnv::txn_checkpoint, and DbMpoolFile::memp_fsync.

The following Java methods are now declared "public int" rather than "public void", and will return `Db.DB_INCOMPLETE` rather than throw an exception: `Db.close()`, `Db.sync()`, and `DbEnv.checkpoint()`.

It is likely that the only change required by any application will be those currently checking for a `DB_INCOMPLETE` return that has been encapsulated in an exception.

Release 3.2: DB_ENV->set_tx_recover

The **info** parameter of the function passed to DB_ENV->set_tx_recover is no longer needed. If your application calls DB_ENV->set_tx_recover, find the callback function referred to by that call and remove the **info** parameter.

In addition, the called function no longer needs to handle Berkeley DB log records, Berkeley DB will handle them internally as well as call the application-specified function. Any handling of Berkeley DB log records in the application's callback function may be removed.

In addition, the callback function will no longer be called with the DB_TXN_FORWARD_ROLL flag specified unless the transaction enclosing the operation successfully committed.

Release 3.2: DB_ENV->set_mutexlocks

Previous Berkeley DB releases included the db_env_set_mutexlocks function, intended for debugging, that allows applications to always obtain requested mutual exclusion mutexes without regard for their availability. This function has been replaced with dbenv_set_mutexlocks, which provides the same functionality on a per-database environment basis. Applications using the old function should be updated to use the new one.

Release 3.2: Java and C++ object reuse

In previous releases of Berkeley DB, Java DbEnv and Db objects, and C++ DbEnv and Db objects could be reused after they were closed, by calling open on them again. This is no longer permitted, and these objects no longer allow any operations after a close. Applications reusing these objects should be modified to create new objects instead.

Release 3.2: Java java.io.FileNotFoundException

The Java DbEnv.remove, Db.remove and Db.rename methods now throw java.io.FileNotFoundException in the case where the named file does not exist. Applications should be modified to catch this exception where appropriate.

Release 3.2: db_dump

In previous releases of Berkeley DB, the db_dump utility dumped Recno access method database keys as numeric strings. For consistency, the db_dump utility has been changed in the 3.2 release to dump record numbers as hex pairs when the data items are being dumped as hex pairs. (See the **-k** and **-p** options to the db_dump utility for more information.) Any applications or scripts post-processing the output of the db_dump utility for Recno databases under these conditions may require modification.

Release 3.2: Upgrade Requirements

Log file formats and the Queue Access Method database formats changed in the Berkeley DB 3.2 release. (The on-disk Queue format changed from version 2 to version 3.) Until the underlying databases are upgraded, the `DB_OLD_VERSION` error.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Chapter 36. Upgrading Berkeley DB 3.2 applications to Berkeley DB 3.3

Release 3.3: introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 3.2 release interfaces to the Berkeley DB 3.3 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 3.3: DB_ENV->set_server

The DB_ENV->set_server method has been deprecated and replaced with the `DB_ENV->set_rpc_server()` method. The DB_ENV->set_server method will be removed in a future release, and so applications using it should convert. The DB_ENV->set_server method can be easily converted to the `DB_ENV->set_rpc_server()` method by changing the name, and specifying a NULL for the added argument, second in the argument list.

Release 3.3: DB->get_type

The DB->get_type() method method can return an error in the Berkeley DB 3.3 release, and so requires an interface change. C and C++ applications calling DB->get_type() should be changed to treat the method's return as an error code, and to pass an additional second argument of type **DBTYPE *** to the method. The additional argument is used as a memory location in which to store the requested information.

Release 3.3: DB->get_byteswapped

The DB->get_byteswapped() method method can return an error in the Berkeley DB 3.3 release, and so requires an interface change. C and C++ applications calling DB->get_byteswapped() should be changed to treat the method's return as an error code, and to pass an additional second argument of type **int *** to the method. The additional argument is used as a memory location in which to store the requested information.

Release 3.3: DB->set_malloc, DB->set_realloc

There are two new methods in the Berkeley DB 3.3 release: DB_ENV->set_alloc(). These functions allow applications to specify a set of allocation functions for the Berkeley DB library to use when allocating memory to be owned by the application and when freeing memory that was originally allocated by the application.

The new methods affect or replace the following historic methods:

DB->set_malloc

The DB->set_malloc method has been replaced in its entirety. Applications using this method should replace the call with a call to DB->set_alloc().

DB->set_realloc

The DB->set_realloc method has been replaced in its entirety. Applications using this method should replace the call with a call to DB->set_alloc().

DB->stat() method

has been replaced. Applications using this method should do as follows: if the argument is NULL, it should simply be removed. If non-NULL, it should be replaced with a call to DB->set_alloc().

lock_stat

The historic **db_malloc** argument to the lock_stat function has been replaced. Applications using this function should do as follows: if the argument is NULL, it should simply be removed. If non-NULL, it should be replaced with a call to DB_ENV->set_alloc().

log_archive

The historic **db_malloc** argument to the log_archive function has been replaced. Applications using this function should do as follows: if the argument is NULL, it should simply be removed. If non-NULL, it should be replaced with a call to DB_ENV->set_alloc().

log_stat

The historic **db_malloc** argument to the log_stat function has been replaced. Applications using this function should do as follows: if the argument is NULL, it should simply be removed. If non-NULL, it should be replaced with a call to DB_ENV->set_alloc().

memp_stat

The historic **db_malloc** argument to the memp_stat function has been replaced. Applications using this function should do as follows: if the argument is NULL, it should simply be removed. If non-NULL, it should be replaced with a call to DB_ENV->set_alloc().

txn_stat

The historic **db_malloc** argument to the txn_stat function has been replaced. Applications using this function should do as follows: if the argument is NULL, it should simply be removed. If non-NULL, it should be replaced with a call to DB_ENV->set_alloc().

One potential incompatibility for historic applications is that the allocation functions for a database environment must now be set before the environment is opened. Historically, Berkeley DB applications could open the environment first, and subsequently call the DB->set_malloc and DB->set_realloc methods; that use is no longer supported.

Release 3.3: DB_LOCK_CONFLICT

The DB_LOCK_CONFLICT flag has been removed from the lock_detect function. Applications specifying the DB_LOCK_CONFLICT flag should simply replace it with a flags argument of 0.

Release 3.3: memp_fget, EIO

Previous releases of Berkeley DB returned the system error EIO when the memp_fget function was called to retrieve a page, the page did not exist, and the DB_MPOOL_CREATE flag was not set. In the 3.3 release, the error DB_PAGE_NOTFOUND is returned instead, to allow applications to distinguish between recoverable and non-recoverable errors. Applications calling the memp_fget function and checking for a return of EIO should check for DB_PAGE_NOTFOUND instead.

Previous releases of Berkeley DB treated filesystem I/O failure (the most common of which the filesystem running out of space), as a fatal error, returning DB_RUNRECOVERY (page 230). When a filesystem failure happens in the 3.3 release Berkeley DB returns the underlying system error (usually EIO), but can continue to run. Applications should abort any enclosing transaction when a recoverable system error occurs in order to recover from the error.

Release 3.3: txn_prepare

An additional argument has been added to the txn_prepare function. If your application calls txn_prepare (that is, is performing two-phase commit using Berkeley DB as a local resource manager), see the section titled *Distributed Transactions* in versions of this book that existed prior to release 4.8.

Release 3.3: --enable-dynamic, --enable-shared

In previous releases, Berkeley DB required separate configuration and builds to create both static and shared libraries. This has changed in the 3.3 release, and Berkeley DB now builds and installs both shared and static versions of the Berkeley DB libraries by default. This change was based on Berkeley DB upgrading to release 1.4 of the GNU Project's Libtool distribution. For this reason, Berkeley DB no longer supports the previous --enable-dynamic and --enable-shared configuration options. Instead, as Berkeley DB now builds both static and shared libraries by default, the useful options are Libtool's --disable-shared and --disable-static options.

Release 3.3: --disable-bigfile

In previous releases, Berkeley DB UNIX used the --disable-bigfile configuration option for systems that could not, for whatever reason, include large file support in a particular Berkeley DB configuration. However, large file support has been integrated into the autoconf configuration tool as of version 2.50. For that reason, Berkeley DB configuration no longer supports --disable-bigfile, the autoconf standard --disable-largefile should be used instead.

Release 3.3: Upgrade Requirements

No database formats or log file formats changed in the Berkeley DB 3.3 release.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Chapter 37. Upgrading Berkeley DB 3.3 applications to Berkeley DB 4.0

Release 4.0: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 3.3 release interfaces to the Berkeley DB 4.0 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.0: db_deadlock

The **-w** option to the db_deadlock utility has been deprecated. Applications can get the functionality of the **-w** option by using the **-t** option with an argument of **.100000**.

Release 4.0: lock_XXX

The C API for the Berkeley DB Locking subsystem was reworked in the 4.0 release as follows:

Historic functional interface	Berkeley DB 4.X method
lock_detect	DB_ENV->lock_detect()
lock_get	DB_ENV->lock_get()
lock_id	DB_ENV->lock_id()
lock_put	DB_ENV->lock_put()
lock_stat	DB_ENV->lock_stat()
lock_vec	DB_ENV->lock_vec()

Applications calling any of these functions should update their calls to use the enclosing DB_ENV handle's method (easily done as the first argument to the existing call is the correct handle to use).

In addition, the DB_ENV->lock_stat() call has been changed in the 4.0 release to take a flags argument. To leave their historic behavior unchanged, applications should add a final argument of 0 to any calls made to DB_ENV->lock_stat().

The C++ and Java APIs for the DbLock::put (DbLock.put) method was reworked in the 4.0 release to make the lock put interface a method of the DB_ENV handle rather than the DbLock handle. Applications calling the DbLock::put or DbLock.put method should update their calls to use the enclosing DB_ENV handle's method (easily done as the first argument to the existing call is the correct handle to use).

Release 4.0: log_XXX

The C API for the Berkeley DB Logging subsystem was reworked in the 4.0 release as follows:

Historic functional interface	Berkeley DB 4.X method
log_archive	DB_ENV->log_archive()
log_file	DB_ENV->log_file()
log_flush	DB_ENV->log_flush()
log_get	DB_ENV->log_cursor()
log_put	DB_ENV->log_put()
log_register	DB_ENV->log_register
log_stat	DB_ENV->log_stat()
log_unregister	DB_ENV->log_unregister

Applications calling any of these functions should update their calls to use the enclosing DB_ENV class handle's method (in all cases other than the log_get call, this is easily done as the first argument to the existing call is the correct handle to use).

Application calls to the historic log_get function must be replaced with the creation of a log file cursor (a DB_LOGC class object), using the DB_ENV->log_cursor() method to retrieve log records and calls to the DB_LOGC->close() method to destroy the cursor. It may also be possible to simplify some applications. In previous releases of Berkeley DB, the DB_CURRENT, DB_NEXT, and DB_PREV flags to the log_get function could not be used by a free-threaded DB_ENV class handle. If their DB_ENV class handle was free-threaded, applications had to create an additional, unique environment handle by separately calling DB_ENV->open(). This is no longer an issue in the log cursor interface, and applications may be able to remove the now unnecessary creation of the additional DB_ENV class object.

Finally, the DB_ENV->log_stat() call has been changed in the 4.0 release to take a flags argument. To leave their historic behavior unchanged, applications should add a final argument of 0 to any calls made to DB_ENV->log_stat().

Release 4.0: memp_XXX

The C API for the Berkeley DB Memory Pool subsystem was reworked in the 4.0 release as follows:

Historic functional interface	Berkeley DB 4.X method
memp_register	DB_ENV->memp_register()
memp_stat	DB_ENV->memp_stat()
memp_sync	DB_ENV->memp_sync()
memp_trickle	DB_ENV->memp_trickle()
memp_fopen	DB_ENV->memp_fcreate()
DB_MPOOL_FINFO: ftype	DB_MPOOLFILE->set_ftype()
DB_MPOOL_FINFO: pgcookie	DB_MPOOLFILE->set_pgcookie()
DB_MPOOL_FINFO: fileid	DB_MPOOLFILE->set_fileid()

Historic functional interface	Berkeley DB 4.X method
DB_MPOOL_FINFO: lsn_offset	DB_MPOOLFILE->set_lsn_offset()
DB_MPOOL_FINFO: clear_len	DB_MPOOLFILE->set_clear_len()
memp_fopen	DB_MPOOLFILE->open()
memp_fclose	DB_MPOOLFILE->close()
memp_fput	DB_MPOOLFILE->put()
memp_fset	DB_MPOOLFILE->set
memp_fsync	DB_MPOOLFILE->sync()

Applications calling any of the memp_register, memp_stat, memp_sync or memp_trickle functions should update those calls to use the enclosing DB_ENV class handle's method (easily done as the first argument to the existing call is the correct DB_ENV class handle).

In addition, the DB_ENV->memp_stat() call has been changed in the 4.0 release to take a flags argument. To leave their historic behavior unchanged, applications should add a final argument of 0 to any calls made to DB_ENV->memp_stat().

Applications calling the memp_fopen function should update those calls as follows: First, acquire a Cache chapter handle using the DB_ENV->memp_fcreate() method. Second, if the DB_MPOOL_FINFO structure reference passed to the memp_fopen function was non-NULL, call the Cache chapter method corresponding to each initialized field in the DB_MPOOL_FINFO structure. Third, call the DB_MPOOLFILE->open() method method to open the underlying file. If the DB_MPOOLFILE->open() method call fails, then DB_MPOOLFILE->close() method must be called to destroy the allocated handle.

Applications calling the memp_fopen, memp_fclose, memp_fput, memp_fset, or memp_fsync functions should update those calls to use the enclosing Cache chapter handle's method. Again, this is easily done as the first argument to the existing call is the correct Cache chapter handle. With one exception, the calling conventions of the old and new interfaces are identical; the one exception is the DB_MPOOLFILE->close() method, which requires an additional flag parameter that should be set to 0.

Release 4.0: txn_XXX

The C API for the Berkeley DB Transaction subsystem was reworked in the 4.0 release as follows:

Historic functional interface	Berkeley DB 4.X method
txn_abort	DB_TXN->abort()
txn_begin	DB_ENV->txn_begin()
txn_checkpoint	DB_ENV->txn_checkpoint()
txn_commit	DB_TXN->commit()
txn_discard	DB_TXN->discard()
txn_id	DB_TXN->id()

Historic functional interface	Berkeley DB 4.X method
txn_prepare	DB_TXN->prepare()
txn_recover	DB_TXN->recover()
txn_stat	DB_TXN->stat()

Applications calling any of these functions should update their calls to use the enclosing DB_ENV class handle's method (easily done as the first argument to the existing call is the correct handle to use).

As a special case, since applications might potentially have many calls to the txn_abort, txn_begin and txn_commit functions, those functions continue to work unchanged in the Berkeley DB 4.0 release.

In addition, the DB_TXN->stat() call has been changed in the 4.0 release to take a flags argument. To leave their historic behavior unchanged, applications should add a final argument of 0 to any calls made to DB_TXN->stat().

Release 4.0: db_env_set_XXX

The db_env_set_region_init function was removed in the 4.0 release and replaced with the DB_REGION_INIT flag to the DB_ENV->set_flags() method. This is an interface change: historically, the db_env_set_region_init function operated on the entire Berkeley DB library, not a single environment. The new method only operates on a single DB_ENV class handle (and any handles created in the scope of that handle). Applications calling the db_env_set_region_init function should update their calls: calls to the historic routine with an argument of 1 (0) are equivalent to calling DB_ENV->set_flags() with the DB_REGION_INIT flag and an argument of 1 (0).

The db_env_set_tas_spins function was removed in the 4.0 release and replaced with the DB_ENV->set_tas_spins method. This is an interface change: historically, the db_env_set_tas_spins function operated on the entire Berkeley DB library, not a single environment. The new method only operates on a single DB_ENV class handle (and any handles created in the scope of that handle). Applications calling the db_env_set_tas_spins function should update their calls: calls to the historic routine are equivalent to calling DB_ENV->set_tas_spins with the same argument. In addition, for consistent behavior, all DB_ENV class handles opened by the application should make the same configuration call, or the value will need to be entered into the environment's DB_CONFIG configuration file (page 128).

Also, three of the standard Berkeley DB debugging interfaces changed in the 4.0 release. It is quite unlikely that Berkeley DB applications use these interfaces.

The DB_ENV->set_mutexlocks method was removed in the 4.0 release and replaced with the DB_NO_LOCKING flag to the DB_ENV->set_flags() method. Applications calling the DB_ENV->set_mutexlocks method should update their calls: calls to the historic routine with an argument of 1 (0) are equivalent to calling DB_NO_LOCKING flag and an argument of 1 (0).

The db_env_set_pageyield function was removed in the 4.0 release and replaced with the DB_YIELDCPU flag to the DB_ENV->set_flags() method. This is an interface change: historically, the db_env_set_pageyield function operated on the entire Berkeley DB library, not a single

environment. The new method only operates on a single DB_ENV class handle (and any handles created in the scope of that handle). Applications calling the db_env_set_pageyield function should update their calls: calls to the historic routine with an argument of 1 (0) are equivalent to calling DB_ENV->set_flags() with the DB_YIELDCPU flag and an argument of 1 (0). In addition, all DB_ENV class handles opened by the application will need to make the same call, or the DB_YIELDCPU flag will need to be entered into the environment's DB_CONFIG configuration file (page 128).

The db_env_set_panicstate function was removed in the 4.0 release, replaced with the DB_PANIC_ENVIRONMENT flags to the DB_ENV->set_flags() method. (The DB_PANIC_ENVIRONMENT flag will cause an environment to panic, affecting all threads of control using that environment. The DB_ENV->set_flags() handle to ignore the current panic state of the environment.) This is an interface change: historically the db_env_set_panicstate function operated on the entire Berkeley DB library, not a single environment. Applications calling the db_env_set_panicstate function should update their calls, replacing the historic call with a call to DB_ENV->set_flags() and the appropriate flag, depending on their usage of the historic interface.

Release 4.0: DB_ENV->set_server

The DB_ENV->set_server() method has been replaced with the DB_ENV->set_rpc_server() method. The DB_ENV->set_server() method can be easily converted to the DB_ENV->set_rpc_server() method by changing the name, and specifying a NULL for the added argument, second in the argument list.

Release 4.0: DB_ENV->set_lk_max

The DB_ENV->set_lk_max method has been deprecated in favor of the DB_ENV->set_lk_max_locks(), DB_ENV->set_lk_max_lockers(), and DB_ENV->set_lk_max_objects() methods. The DB_ENV->set_lk_max method continues to be available, but is no longer documented and is expected to be removed in a future release.

Release 4.0: DB_ENV->lock_id_free

A new locker ID related API, the DB_ENV->lock_id_free() method, was added to Berkeley DB 4.0 release. Applications using the DB_ENV->lock_id() method to allocate locker IDs may want to update their applications to free the locker ID when it is no longer needed.

Release 4.0: Java CLASSPATH environment variable

The Berkeley DB Java class files are now packaged as jar files. In the 4.0 release, the CLASSPATH environment variable must change to include at least the db.jar file. It can optionally include the dbexamples.jar file if you want to run the examples. For example, on UNIX:

```
export CLASSPATH="/usr/local/BerkeleyDB.4.8/lib/db.jar:/usr/local/BerkeleyDB.4.8/lib/dbexample
```

For example, on Windows:

```
set CLASSPATH="D:\db\build_windows\Release\db.jar;D:\db\build_windows\Release\dbexamples.jar"
```

For more information on Java configuration, please see Java configuration (page 84) and Building the Java API (page 314).

Release 4.0: C++ ostream objects

In the 4.0 release, the Berkeley DB C++ API has been changed to use the ISO standard C++ API in preference to the older, less portable interfaces, where available. This means the Berkeley DB methods that used to take an ostream object as a parameter now expect a std::ostream. Specifically, the following methods have changed:

```
DbEnv::set_error_stream
Db::set_error_stream
Db::verify
```

On many platforms, the old and the new C++ styles are interchangeable; on some platforms (notably Windows systems), they are incompatible. If your code uses these methods and you have trouble with the 4.0 release, you should update code that looks like this:

```
#include <iostream.h>
#include <db_cxx.h>

void foo(Db db) {
 db.set_error_stream(&cerr);
}
```

to look like this:

```
#include <iostream>
#include <db_cxx.h>

using std::cerr;

void foo(Db db) {
 db.set_error_stream(&cerr);
}
```

Release 4.0: application-specific recovery

If you have created your own logging and recovery routines, you may need to upgrade them to the Berkeley DB 4.0 release.

First, you should regenerate your logging, print, read and the other automatically generated routines, using the dist/gen_rec.awk tool included in the Berkeley DB distribution.

Next, compare the template file code generated by the gen_rec.awk tool against the code generated by the last release in which you built a template file. Any changes in the templates should be incorporated into the recovery routines you have written.

Third, if your recovery functions refer to DB_TXN_FORWARD_ROLL (that is, your code checks for that particular operation code), you should replace it with DB_REDO(op) which compares

the operation code to both DB_TXN_FORWARD_ROLL and DB_TXN_APPLY. (DB_TXN_APPLY is a potential value for the operation code as of the 4.0 release.)

Finally, if you have created your own logging and recovery routines, we recommend you contact us and ask us to review those routines for you.

Release 4.0: Upgrade Requirements

The log file format changed in the Berkeley DB 4.0 release. No database formats changed in the Berkeley DB 4.0 release.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

4.0.14 Change Log

Major New Features:

1. Group commit. [#42]
2. Single-master replication. [#44]
3. Support for VxWorks AE; Vxworks support certified by WindRiver Systems Inc. [#4401]

General Environment Changes:

1. The db_env_set_pageyield interface has been replaced by a new flag (DB_YIELDCPU) for the DB_ENV->set_flags interface.
2. The db_env_set_panicstate interface has been replaced by a new flag (DB_PANIC_STATE) for the DB_ENV->set_flags interface.
3. The db_env_set_region_init interface has been replaced by a new flag (DB_REGION_INIT) for the DB_ENV->set_flags interface.
4. The db_env_set_tas_spins interface has been replaced by the DB_ENV->set_tas_spins method.
5. The DB_ENV->set_mutexlocks interface has been replaced by a new flag (DB_NOLOCKING) for the DB_ENV->set_flags interface.
6. Fix a bug where input values from the DB_CONFIG file could overflow.
7. The C API lock, log, memory pool and transaction interfaces have been converted to method based interfaces; see the Upgrade documentation for specific details. [#920]
8. Fix a bug in which some DB_ENV configuration information could be lost by a failed DB_ENV->open command. [#4608]
9. Fix a bug where Berkeley DB could fail if the application attempted to allocate new database pages while the system was unable to write new log file buffers. [#4928]

General Access Method Changes:

1. Add a new flag (DB_GET_BOTH_RANGE) that adds support for range searches within sorted duplicate data sets. [#3378]

2. Fix a bug in which the DB->get or DB->pget methods, when used with secondary indices, could incorrectly leave an internally-created database cursor open. [#4465]

3. The DB->set_alloc method can no longer be called when the database is part of a database environment. [#4599]

Btree Access Method Changes:

1. Fix a bug where a lock could be leaked when a thread calling DB->stat on a Btree database was selected to resolve a deadlock. [#4509]

Hash Access Method Changes:

1. Fix a bug where bulk return using the MULTIPLE_KEY flag on a Hash database would only return entries from a single bucket. [#4313]

Queue Access Method Changes:

1. Delete extent files whenever the leading record is deleted, instead of only when a DB_CONSUME operation was performed. [#4307]

Recno Access Method Changes:

1. Fix a bug where the delete of a record in a Recno database could leak a lock in non-transactional applications. [#4351]

2. Fix a bug where the DB_THREAD flag combined with a backing source file could cause an infinite loop. [#4581]

C++ API Changes:

Java API Changes:

1. Added implementation of DbEnv.lock_vec for Java. [#4094] Added some minimal protection so that the same Java Dbt cannot be used twice in the same API call, this will often catch multithreading programming errors with Dbts. [#4094]

2. Fix a bug in which a Db.put call with the Db.DB_APPEND would fail to correctly return the newly put record's record number. [#4527]

3. Fixed problems occurring in multithreaded java apps that use callbacks. [#4467]

Tcl API Changes:

1. Fix a bug in which large integers could be handled incorrectly by the Tcl interface on 64-bit machines. [#4371]

RPC Client/Server Changes:

1. The DB_ENV->set_server interface has been removed.

XA Resource Manager Changes:

Locking Subsystem Changes:

1. The C++ (Java) API DbLock::put (DbLock.put) method has been changed to be a method off the DbEnv handle rather than the DbLock handle.

2. Locker IDs may now wrap-around. [#864]

3. Explicitly allocated locker IDs must now be freed. [#864]

4. Add per-environment, per-lock and per-transaction interfaces to support timeout based lock requests and "deadlock" detection. [#1855]

5. Add support for interrupting a waiting locker. [#1976]

6. Implemented DbEnv.lock_vec for Java. [#4094]

Logging Subsystem Changes:

1. Fix a bug where the size of a log file could not be set to the default value. [#4567]

2. Fix a bug where specifying a non-default log file size could cause other processes to be unable to join the environment and read its log files. [#4567]

3. Fix a bug where Berkeley DB could keep open file descriptors to log files returned by the DB_ENV->log_archive method (or the db_archive utility), making it impossible to move or remove them on Windows systems. [#3969]

4. Replace the log_get interface with a cursor into the log file. [#0043]

Memory Pool Subsystem Changes:

1. Add the DB_ODDFILESIZE flag to the DB_MPOOLFILE->open method supporting files not a multiple of the underlying page size in length.

2. Convert memp_XXX functional interfaces to a set of methods, either base methods off the DB_ENV handle or methods off of a DB_MPOOLFILE handle. [#920]

3. Add the DB_ODDFILESIZE flag to the DB_MPOOLFILE->open method supporting files not a multiple of the underlying page size in length.

4. Fix a bug where threads of control could deadlock opening a database environment with multiple memory pool caches. [#4696]

5. Fix a bug where the space needed for per-file memory pool statistics was incorrectly calculated. [#4772]

Transaction Subsystem Changes:

1. Transaction IDs may now wrap-around. [#864]

2. Release read locks before performing logging operations at commit. [#4219]

Utility Changes:

1. Fix a bug in which the db_dump utility would incorrectly attach to transaction, locking, or logging regions when salvaging, and thus could not be used to salvage databases in environments where these regions were present. [#4305]

2. Fix a bug in which the DB salvager could produce incorrectly formatted output for certain classes of corrupt database. [#4305]

3. Fix a bug in which the DB salvager could incorrectly salvage files containing multiple databases. [#4305]

4. Fix a bug where unprintable characters in subdatabase names could cause a dump of a database that could not then be loaded. [#4688]

5. Increase the size of the cache created by the db_stat and db_verify utilities to avoid failure on large databases. [#4688] [#4787]

6. Fix a bug in which a database verification performed with the DB_ORDERCHKONLY flag could fail incorrectly. [#4757]

7. Fix a bug which caused db_stat to display incorrect information about GB size caches. [#4812]

Database or Log File On-Disk Format Changes:

1. The on-disk log format changed.

Configuration, Documentation, Portability and Build Changes:

1. Fix a bug where Win9X systems region names could collide.

2. Fix a bug where configuring Berkeley DB to build the C++ API without also configuring for a shared library build would fail to build the C++ library. [#4343]

3. Change Berkeley DB installation to not strip binaries if --enable-debug was specified as a configuration option. [#4318]

4. Add the -pthread flag to AIX, FreeBSD and OSF/1 library loads. [#4350]

5. Fix a bug where the Berkeley DB 1.85 compatibility API failed to load in the 3.3.11 release. [#4368]

6. Port the Berkeley DB utility programs to the VxWorks environment. [#4378]

7. Made change to configuration so that dynamic libraries link correctly when C++ is used on AIX. [#4381]

8. Fix a variety of problems that prevented the Berkeley DB source tree from building on systems without ANSI C compiler support (for example, SunOS 4.X). [#4398]

9. Added missing DbMultiple*Iterator Java files to Makefile.in. [#4404]

10. Fix a bug that could prevent the db_dump185 utility from dumping Berkeley DB version 1.86 hash databases. [#4418]

11. Reduce the number of calls setting the errno value, to improve performance on Windows/NT in MT environments. [#4432]

12. Fix for Darwin (and probably some other) OS's that were getting 'yes' or other garbage in generated makefiles in place of a shared library name. [#4453]

13. C++: Remove inlining for constructor of tmpString internal class. This fixes warnings on Solaris profiling builds. [#4473]

14. DB now restarts system calls that are interrupted by signals. [#4480]

15. Fixed warnings for compiling Java native code on Solaris and OSF/1. [#4571]

16. Added better configuration for Java on Tru64 (OSF/1), Solaris,

17. Java files are now built as jar files. Berkeley DB classes are put into db.jar (which is an installed file on UNIX) and examples are put into dbexamples.jar. The classes directory is now a subdirectory of the build directory, rather than in java/classes. [#4575]

18. Support Cygwin installation process. [#4611]

19. Correct the Java secondary_key_create method signature. [#4777]

20. Export additional Berkeley DB interfaces on Windows to support application-specific logging and recovery. [#4827]

21. Always complain when using version 2.96 of the gcc compiler. [#4878]

22. Add compile and load-time flags to configure for threads on UnixWare and OpenUNIX. [#4552] [#4950]

Chapter 38. Upgrading Berkeley DB 4.0 applications to Berkeley DB 4.1

Release 4.1: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.0 release interfaces to the Berkeley DB 4.1 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.1: DB_EXCL

The DB_EXCL flag to the DB->open() method now works for subdatabases as well as physical files, and it is now possible to use the DB_EXCL flag to check for the previous existence of subdatabases.

Release 4.1: DB->associate, DB->open, DB->remove, DB->rename

Historic releases of Berkeley DB transaction-protected the DB->open(), DB->remove(), and DB->rename() methods, but did it in an implicit way, that is, applications did not specify the TXN handles associated with the operations. This approach had a number of problems, the most significant of which was there was no way to group operations that included database creation, removal or rename. For example, applications wanting to maintain a list of the databases in an environment in a well-known database had no way to update the well-known database and create a database within a single transaction, and so there was no way to guarantee the list of databases was correct for the environment after system or application failure. Another example might be the creation of both a primary database and a database intended to serve as a secondary index, where again there was no way to group the creation of both databases in a single atomic operation.

In the 4.1 release of Berkeley DB, this is no longer the case. The DB->open() and DB->associate() methods now take a TXN handle returned by DB_ENV->txn_begin() as an optional argument. New DB_ENV->dbremove() and DB_ENV->dbrename() methods taking a TXN handle as an optional argument have been added.

To upgrade, applications must add a TXN parameter in the appropriate location for the DB->open() method calls, and the DB->associate() method calls (in both cases, the second argument for the C API, the first for the C++ or Java APIs).

Applications wanting to transaction-protect their DB->open() and DB->associate() method calls can add a NULL TXN argument and specify the DB_AUTO_COMMIT flag to the two calls, which wraps the operation in an internal Berkeley DB transaction. Applications wanting to transaction-protect the remove and rename operations must rewrite their calls to the DB->remove() and DB->rename() methods to be, instead, calls to the new DB_ENV->dbremove() and DB_ENV->dbrename() methods. Applications not wanting to transaction-protect any of the operations can add a NULL argument to their DB->open() and DB->associate() method calls and require no further changes.

For example, an application currently opening and closing a database as follows:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = db_create(&dbp, dbenv, 0)) != 0)
 goto err_handler;

if ((ret = dbp->open(dbp, "file", NULL, DB_BTREE, DB_CREATE, 0664)) != 0) {
 (void)dbp->close(dbp);
 goto err_handler;
}
```

could transaction-protect the DB->open() call as follows:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = db_create(&dbp, dbenv, 0)) != 0)
 goto err_handler;

if ((ret = dbp->open(dbp,
    NULL, "file", NULL, DB_BTREE, DB_CREATE | DB_AUTO_COMMIT, 0664)) != 0) {
 (void)dbp->close(dbp);
 goto err_handler;
}
```

An application currently removing a database as follows:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = db_create(&dbp, dbenv, 0)) != 0)
 goto err_handler;

if ((ret = dbp->remove(dbp, "file", NULL, 0)) != 0)
 goto err_handler;
```

could transaction-protect the database removal as follows:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret =
    dbenv->dbremove(dbenv, NULL, "file", NULL, DB_AUTO_COMMIT)) != 0)
 goto err_handler;
```

An application currently renaming a database as follows:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = db_create(&dbp, dbenv, 0)) != 0)
 goto err_handler;

if ((ret = dbp->rename(dbp, "file", NULL, "newname", 0)) != 0)
 goto err_handler;
```

could transaction-protect the database renaming as follows:

```
DB *dbp;
DB_ENV *dbenv;
int ret;

if ((ret = dbenv->dbrename(
    dbenv, NULL, "file", NULL, "newname", DB_AUTO_COMMIT)) != 0)
 goto err_handler;
```

These examples are the simplest possible translation, and will result in behavior matching that of previous releases. For further discussion on how to transaction-protect DB->open() method calls, see Opening the databases (page 153).

DB handles that will later be used for transaction-protected operations must be opened within a transaction. Specifying a transaction handle to operations using handles not opened within a transaction will return an error. Similarly, not specifying a transaction handle to operations using handles that were opened within a transaction will also return an error.

Release 4.1: DB_ENV->log_register

The DB_ENV->log_register and DB_ENV->log_unregister interfaces were removed from the Berkeley DB 4.1 release. It is very unlikely application programs used these interfaces. If your application used these interfaces, please contact us for help in upgrading.

Release 4.1: st_flushcommit

The DB_ENV->log_stat "st_flushcommits" statistic has been removed from Berkeley DB, as it is now the same as the "st_scount" statistic. Any application using the "st_flushcommits" statistic should remove it, or replace it with the "st_count" statistic.

Release 4.1: DB_CHECKPOINT, DB_CURLSN

The DB_CHECKPOINT flag has been removed from the DB_LOGC->get() and DB_ENV->log_put() methods. It is very unlikely application programs used this flag. If your application used this flag, please contact us for help in upgrading.

The DB_CURLSN flag has been removed from the DB_ENV->log_put() method. It is very unlikely application programs used this flag. If your application used this flag, please contact us for help in upgrading.

Release 4.1: DB_INCOMPLETE

The DB_INCOMPLETE error has been removed from the 4.1 release, and is no longer returned by the Berkeley DB library. Applications no longer need to check for this error return, as the underlying Berkeley DB interfaces that could historically fail to checkpoint or flush the cache and return this error can no longer fail for that reason. Applications should remove all uses of DB_INCOMPLETE.

Additionally, the DbEnv.checkpoint and Db.sync methods have been changed from returning int to returning void.

Release 4.1: DB_ENV->memp_sync

Historical documentation for the DB_ENV->memp_sync() method stated:
In addition, if DB_ENV->memp_sync() returns success, the value of **lsn** will be overwritten with the largest log sequence number from any page that was written by DB_ENV->memp_sync() to satisfy this request.

This functionality was never correctly implemented, and has been removed in the Berkeley DB 4.1 release. It is very unlikely application programs used this information. If your application used this information, please contact us for help in upgrading.

Release 4.1: DB->stat.hash_nelem

The **hash_nelem** field of the DB->stat() method for Hash databases has been removed from the 4.1 release, this information is no longer available to applications.

Release 4.1: Java exceptions

The Java DbEnv constructor is now marked with "throws DbException". This means applications must construct DbEnv objects in a context where DbException throwables are handled (either in a try/catch block or in a method that propagates the exception up the stack). Note that previous versions of the Berkeley DB Java API could throw this exception from the constructor but it was not marked.

Release 4.1: C++ exceptions

With default flags, the C++ DbEnv and Db classes can throw exceptions from their constructors. For example, this can happen if invalid parameters are passed in or the underlying C structures could not be created. If the objects are created in an environment that is not configured for exceptions (that is, the DB_CXX_NO_EXCEPTIONS flag is specified), errors from the constructor will be returned when the handle's open method is called.

In addition, the behavior of the DbEnv and Db destructors has changed to simplify exception handling in applications. The destructors will now close the handle if the handle's close method

was not called prior to the object being destroyed. The return value of the call is discarded, and no exceptions will be thrown. Applications should call the close method in normal situations so any errors while closing can be handled by the application.

This change allows applications to be structured as follows:

```
try {
 DbEnv env(0);
 env.open(/* ... */);
 Db db(&env, 0);
 db.open(/* ... */);
 /* ... */
 db.close(0);
 env.close(0);
} catch (DbException &dbe) {
 // Handle the exception, the handles have already been closed.
}
```

Release 4.1: Application-specific logging and recovery

The application-specific logging and recovery tools and interfaces have been reworked in the 4.1 release to make it simpler for applications to use Berkeley DB to support their own logging and recovery of non-Berkeley DB objects. Specifically, the DB_ENV->set_recovery_init and DB_ENV->set_tx_recover interfaces have been removed, replaced by DB_ENV->set_app_dispatch(). Applications using either of the removed interfaces should be updated to call DB_ENV->set_app_dispatch(). For more information see Introduction to application specific logging and recovery (page 220) and the DB_ENV->set_app_dispatch() documentation.

Release 4.1: Upgrade Requirements

The log file format changed in the Berkeley DB 4.1 release.

All of the access method database formats changed in the Berkeley DB 4.1 release (Btree/Recno: version 8 to version 9, Hash: version 7 to version 8, and Queue: version 3 to version 4). **The format changes are entirely backward-compatible, and no database upgrades are needed.** Note that databases created using the 4.1 release may not be usable with earlier Berkeley DB releases.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.1.24 and 4.1.25 Change Log

Database or Log File On-Disk Format Changes:

1. All of the access method database formats changed in the Berkeley DB 4.1 release (Btree/Recno: version 8 to version 9, Hash: version 7 to version 8, and Queue: version 3 to

version 4). *The format changes are entirely backward-compatible, and no database upgrades are needed.*

Major New Features:

1. Berkeley DB now includes support for database encryption using the AES encryption standard. [#1797]
2. Berkeley DB now includes support for database page checksums to allow detection of database corruption during I/O. [#1797]
3. The shared memory buffer pool code base was substantially reworked in the 4.1 release to improve concurrent throughput. [#4655]

General Environment Changes:

1. Allow applications to specify transaction handles to the DB->open method call, so database creation can be grouped with other Berkeley DB calls in a single transaction. [#4257]
2. Add the DB_ENV->remove and DB_ENV->rename method calls that support transactional protection of database removal and renaming. [#4257]
3. Add the DB_ENV->set_flags flags DB_DIRECT_DB and DB_DIRECT_LOG, which disable the system's buffer cache where possible. [#4526]
4. Unlock the pthread mutex if pthread_cond_wait() returns an error. [#4872]
5. Fix a memory leak caused by running recovery. [#4913]
6. Fix a bug in which closing an environment with open database handles could result in application crashes. [#4991]
7. Fix a bug where DB_CONFIG files were ignored if the database environment defaulted to the application's current working directory. [#5265]
8. Fix a bug where transaction abort or commit could fail to destroy the handle. [#5633]
9. Fix a set of bugs where the Berkeley DB API could return DB_RUNRECOVERY without panic-ing the database environment itself or calling the application's panic-callback function. [#5743]
10. Fix a bug in where DB=>rename and DB->remove method calls could leak a transaction and its locks. [#5824]
11. Fix a bug where recovery feedback could return values greater than 100. [#6193]
12. Fix a bug where a page allocated by a transaction, eventually aborted because of application or system failure, could appear twice in the free list, if catastrophic recovery was performed. [#6222]
13. Add a new flag, DB_AUTO_COMMIT, that wraps all database modification operations inside a transaction, to the DB_ENV->set_flags method. [#6395]

14. Fix a bug where recovery could fail when upgrading between releases. [#6372]

15. Fix a recovery bug where pages that were repeatedly freed and allocated could be lost. [#6479] [#6501]

16. Change DB_CONFIG reading to handle non-<newline> terminated last line. [#6490]

General Access Method Changes:

1. Allow applications to specify transaction handles to the DB->associate method call, so secondary index creation can be grouped with other Berkeley DB calls in a single transaction. [#4185]

2. Add a new flag, DB_AUTO_COMMIT, that wraps single database operations inside a transaction. This flag is supported by the DB->del, DB->open, DB->put, DB->truncate,DB_ENV->remove, and DB_ENV->rename methods. [#4257]

3. The DB_EXCL DB->open method flag has been enhanced to work on subdatabases. [#4257]

4. Fix a bug in which a DB->put(DB_APPEND) could result in leaked memory or a corruption in the returned record number. [#5002]

5. Fix a bug in the database salvage code that could leave pages pinned in the cache. [#5037]

6. Add a flag to the DB->verify method to output salvaged key/data pairs in printable characters. [#5037]

7. Fix a bug in which DB->verify() might continue and report extraneous database corruption after a fatal error. [#5131]

8. Fix a bug where calling the DB->stat method before the DB->open method could drop core. [#5190]

9. Fix a bug in which a DB->get, DBcursor->c_get, or DBcursor->c_pget on a secondary index, in the Concurrent Data Store product, could result in a deadlock. [#5192]

10. Fix a bug in which DB->verify() could correctly report errors but still return success. [#5297]

11. Add support for the DB->set_cache_priority interface, that allows applications to set the underlying cache priority for their database files. [#5375]

12. Fix a bug where calling DBcursor->c_pget with a database that is not a secondary index would drop core. [#5391]

13. Fix a bug where a bug in the DB->truncate method could cause recovery to fail. [#5679]

14. Fix a bug where DB_GET_RECNO would fail if specified to a secondary index. [#5811]

15. Fix a bug where building a secondary index for an existing primary database could fail in Concurrent Data Store environments. [#5811]

16. Fix a bug where the DB->rename method could fail, causing a problem during recovery. [#5893]

17. Fix a bug in which a DB->get or DB->pget call on a secondary index could fail when done with a handle shared among multiple threads. [#5899]

18. Fix a bug in which a DB->put operation on a database with off-page duplicates could leak a duplicate cursor, thereby preventing transactions being able to commit. [#5936]

19. Fix a bug where overflow page reference counts were not properly maintained when databases were truncated. [#6168]

20. Fix a bug where the bulk get APIs could allocate large amounts of heap memory. [#6439] [#6520]

Btree Access Method Changes:

1. Fix a bug that prevented loads of sorted data, with duplicates at the end of the tree, from creating compact trees. [#4926]

2. No longer return a copy of the key if the DB_GET_BOTH or DB_GET_BOTH_RANGE flags are specified. [#4470]

3. Fix a bug where the fast-search code could hold an unlocked reference to a page, which could lead to recovery failure. [#5518]

4. Fix a bug where some cursor operations on a database, for which the bt_minkey size had been specified, could fail to use the correct overflow key/data item size. [#6183]

5. Fix a bug where the recovery of an aborted transaction that did a reverse Btree split might leave a page in an inconsistent state. [#6393]

Hash Access Method Changes:

1. Fix bugs that could cause hash recovery to drop core. [#4978]

2. Use access method flags instead of interface flags to check for read-only access to a hash database with an application-specified hash function. [#5121]

3. Fix a bug where a hash database allocation of a new set of buckets may be improperly recovered by catastrophic recovery if the transaction is split across log files and the beginning segment of the transaction is not included in the set of logs to be recovered. [#5942]

4. Fix a bug where aborting particular hash allocations could lead to a database on which the verifier would loop infinitely. [#5966]

5. Fix a bug where a memory allocation failure could result in a system hang. [#5988]

6. Remove nelem from the Hash access method statistics (the value was incorrect once items had been added or removed from the database). [#6101]

7. Fix a bug where a page allocated by an aborted transaction might not be placed on the free list by recovery, if the file holding the page was created as part of recovery, and a later page was part of a hash bucket allocation. [#6184]

8. Fix a bug where allocated pages could be improperly recovered on systems that require explicit zero-ing of filesystem pages. [#6534]

Queue Access Method Changes:

1. No longer return a copy of the key if the DB_SET_RANGE flag is specified. [#4470]

2. Fix a bug where DBcursor->c_get (with DB_MULTIPLE or DB_MULTIPLE_KEY specified) could fail on a Queue database if the record numbers had wrapped. [#6397]

Recno Access Method Changes:

1. No longer return a copy of the key if the DB_GET_BOTH or DB_GET_BOTH_RANGE flags are specified. [#4470]

2. Fix a bug where non-transactional locking applications could leak locks when modifying Recno databases. [#5766]

3. Fix a bug where DBcursor->c_get with the DB_GET_RECNO flag would panic the environment if the cursor was uninitialized. [#5935]

4. Fix a bug where deleting pages from a three-level Recno tree could cause the database environment to panic. [#6232]

C++-specific API Changes:

1. C++ DbLock::put is replaced by DbEnv::lock_put to match the C and Java API change in Release 4.0. [#5170]

2. Declared destructors and methods within Db and DbEnv classes to be virtual, making subclassing safer. [#5264]

3. Fixed a bug where Dbt objects with no flags set would not be filled with data by some operations. [#5706]

4. Added DbDeadlockException, DbRunRecoveryException, and DbLockNotGrantedException classes to C++, and throw them accordingly. [#6134]

5. Added C++ methods to support remaining conversions between C++ classes and C structs where appropriate. In particular, DbTxn/DB_TXN conversions and DbMpoolFile/DB_MPOOLFILE were added. [#6278]

6. Fix a bug in DbEnv::~DbEnv() that could cause memory corruption if a DbEnv was deleted without being closed. [#6342]

7. Reordered C++ class declarations to avoid a GCC g++ warning about function inlining. [#6406]

8. Fix a bug in the DbEnv destructor that could cause memory corruption when an environment was destroyed without closing first. [#6342]

9. Change DbEnv and Db destructor behavior to close the handle if it was not already closed. [#6342]

Java-specific API Changes:

1. Added check for system property "sleepycat.Berkeley DB.libfile" that can be used to specify a complete pathname for the JNI shared library. This is needed as a workaround on Mac OS X, where libtool cannot currently create a library with a .jnilib extension which is what the current JDK expects by default. [#5664]

2. Fixed handling of JVM out of memory conditions, when some JNI methods return NULL. When the JVM runs out of memory, calls should consistently fail with OutOfMemoryErrors. [#5995]

3. Added Dbt.get_object and Dbt.set_object convenience routines to the Java API to make using serialization easier. [#6113]

4. Fixed a bug that prevented Java's Db.set_feedback from working, fixed document for Java's Db.set_feedback, some callback methods were misnamed. [#6137]

5. Fix a NullPointerException in Db.finalize() if the database had been closed. [#6504]

6. Marked DbEnv constructor with "throws DbException". [#6342]

Tcl-specific API Changes:

None.

RPC-specific Client/Server Changes:

1. Fix a bug where Db and DbEnv handles were not thread-safe. [#6102]

Replication Changes:

1. A large number of replication bugs were fixed in this release. The replication support is now believed to be production quality.

2. Add the DB_ENV->set_rep_limit interface, allowing applications to limit the data sent in response to a single DB_ENV->rep_process_message call. [#5999]

3. Add the DB_ENV->set_rep_stat interface, returning information from the replication subsystem [#5919]

XA Resource Manager Changes:

1. Added support for multithreaded XA. Environments can now have multiple XA transactions active. db_env_xa_attach() can be used to get a DB_TXN that corresponds to the XA transaction in the current thread. [#5049]

2. Added a com.sleepycat.Berkeley DB.xa package that implements J2EE support for XA. This includes new DbXAResource, DbXid classes that implement the XAResource and Xid interfaces. [#5049]

3. Fix a bug where aborting a prepared transaction after recovery may fail. [#6383]

4. Fix a bug where recovery might fail if a prepared transaction had previously extended the size of a file and then was aborted. [#6387]

5. Fix a bug where if the commit of a prepared transaction fails the transaction would be aborted. [#6389]

Locking Subsystem Changes:

1. Fix a bug where lock counts were incorrect if a lock request returned DB_LOCK_NOTGRANTED or an error occurred. [#4923]

2. Fix a bug where lock downgrades were counted as releases, so the lock release statistics could be wrong. [#5762]

3. Fix a bug where the lock and transaction timeout values could not be reset by threads of control joining Berkeley DB database environments. [#5996]

4. Fix a bug where applications using lock and/or transaction timeouts could hit a race condition that would lead to a segmentation fault. [#6061]

Logging Subsystem Changes:

1. DB_ENV->log_register and DB_ENV->log_unregister have been removed from the interface. [#0046]

2. Fix a bug where creating a database environment with a nonexistent logging directory could drop core. [#5833]

3. Add support allowing applications to change the log file size in existing database environments. [#4875]

4. Fix a bug where a write error on a log record spanning a buffer could cause transaction abort to fail and the database environment to panic. [#5830]

Memory Pool Subsystem Changes:

1. The DB_INCOMPLETE error has been removed, as cache flushing can no longer return without completing. [#4655]

2. Fix a bug where Berkeley DB might refuse to open a file if the open was attempted while another thread was writing a large buffer. [#4885]

3. Prefer clean buffers to dirty buffers when selecting a buffer for eviction. [#4934]

4. Fix a bug where transaction checkpoint might miss flushing a buffer to disk. [#5033]

5. Fix a bug where Berkeley DB applications could run out of file descriptors. [#5535]

6. Fix bugs where Berkeley DB could self-deadlock on systems requiring mutex resource reclamation after application failure. [#5722] [#6523]

Transaction Subsystem Changes:

1. Go back only one checkpoint, not two, when performing normal recovery. [#4284]

2. Fix a bug where an abort of a transaction could fail if there was no disk space for the log. [#5740]

3. Fix a bug where the checkpoint log-sequence-number could reference a nonexistent log record. [#5789]

4. Fix a bug where subtransactions which allocated pages from the filesystem and subsequently aborted could cause other pages allocated by sibling transactions to not be freed if the parent transaction then aborted. [#5903]

5. Fix a bug where transactions doing multiple updates to a queue database which spanned a checkpoint could be improperly handled by recovery. [#5898]

Utility Changes:

1. Fix a bug where the -p option could not be specified with the -R or -r options. [#5037]

2. The utilities were modified to correctly size their private caches in order to handle databases with large page sizes. [#5055]

3. Fix a bug in which utilities run with the -N option would fail to ignore the environment's panic flag. [#5082]

4. Fix a bug where invalid log records could cause db_printlog to drop core. [#5173]

5. Add a new option to the db_verify utility to support verification of files that include databases having non-standard sorting or hash functions. [#5237]

Configuration, Documentation, Portability and Build Changes:

1. Replace test-and-set mutexes on Windows with a new mutex implementation that signals an event to wake blocked threads. [#4413]

2. Support configuration of POSIX pthread mutexes on systems where the pthread mutexes do not support inter-process locks. [#4942]

3. Add mutex support for the ARM architecture using the gcc compiler. [#5018]

4. On Windows NT/2000/XP, switched to atomic seek-and-read/write operations to improve performance of concurrent reads [#0654].

5. Support cross-compilation using the GNU compiler tool chain. [#4558]

6. Fix a bug where libraries were always installed read-only. [#5096]

7. Fix a bug where temporary files on VxWorks could fail. [#5160]

8. Fix a bug where Berkeley DB did not install correctly if the system cp utility did not support the -f option. [#5111]

9. Correct the documentation for the Queue access method statistics field qs_cur_recno to be the "Next available record number". [#5190]

10. Fix a bug where file rename could fail on Windows/9X. [#5223]

11. Removed support for Microsoft Visual Studio 5.0 [#5231]

12. Switched to using HANDLEs for all I/O operations on Windows to overcome a hard limit of 2048 open file descriptors in Microsoft's C runtime library. [#5249]

13. Fix a bug where Berkeley DB error message routines could drop core on the PowerPC and UltraSPARC architectures. [#5331]

14. Rename OSTREAMCLASS to __DB_OSTREAMCLASS in db_cxx.h to avoid stepping on application name space. [#5402]

15. Support Linux on the S/390 architecture. [#5608]

16. Work around a bug in Solaris where the pthread_cond_wait call could return because a signal was delivered to the application. [#5640]

17. Fix build line for loadable libraries to include -module to support Mac OS X. [#5664]

18. Fix a bug in the PPC mutex support for the Mac OS X system. [#5781]

19. Added support for Java on Mac OS X. A workaround on the Java command line is currently necessary; it is documented. [#5664]

20. Added support for Tcl on Mac OS X. [#5664]

21. Update Windows build instructions to cover Visual C++ .NET. [#5684]

22. AIX configuration changes for building on AIX 4.3.3 and 5 with both standard and Visual Age compilers. [#5779]

23. Add a new UNIX configuration argument, --with-mutex=MUTEX, to allow applications to select a mutex implementation. [#6040]

24. Changed libtool and configure so we can now correctly build and install Tcl and Java loadable shared libraries that work on Mac OS X. [#6117]

25. Fix mutex alignment problems on historic HP-UX releases that could make multiprocess applications fail. [#6250]

26. Installed static .a archives on Mac OS X need to be built with the ranlib -c option so linked applications will not see undefined __db_jump errors. [#6215]

27. Upgrade pthread and mmap support in the uClibc library to support Berkeley DB. [#6268]

28. Fixed error in determining include directories during configuration for --enable-java. The error can cause compilation errors on certain systems with newer versions of gcc. [#6445]

Berkeley DB 4.1.25 Change Log

Berkeley DB version 4.1.25 is version 4.1.24 with all public patches applied. There were no public interface changes or new features.

Chapter 39. Upgrading Berkeley DB 4.1 applications to Berkeley DB 4.2

Release 4.2: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.1 release interfaces to the Berkeley DB 4.2 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.2: Java

There are a number of major changes to the Java support in Berkeley DB in this release. Despite that we have tried to make this a bridge release, a release where we don't require you to change anything. We've done this using the standard approach to deprecation in Java. If you do not compile with deprecation warnings on, your existing sources should work with this new release with only minor changes despite the large number of changes. Expect that in a future release we will remove all the deprecated API and only support the new API names.

This is a list of areas where we have broken compatibility with the 4.1 release. In most cases it is a name change in an interface class.

- **DbAppDispatch.app_dispatch(DbEnv,Dbt,DbLsn,int)**

  is now: **DbAppDispatch.appDispatch(DbEnv,Dbt,DbLsn,int)**

- **DbAppendRecno.db_append_recno(Db,Dbt,int)**

  is now: **DbAppendRecno.dbAppendRecno(Db,Dbt,int)**

- **DbBtreeCompare.bt_compare(Db,Dbt,Dbt)**

  is now: **DbBtreeCompare.compare(Db,Dbt,Dbt)**

- **DbBtreeCompare.dup_compare(Db,Dbt,Dbt)**

  is now: **DbBtreeCompare.compareDuplicates(Db,Dbt,Dbt)**

- **DbBtreePrefix.bt_prefix(Db,Dbt,Dbt)**

  is now: **DbBtreePrefix.prefix(Db,Dbt,Dbt)**

- **DbSecondaryKeyCreate.secondary_key_create(Db,Dbt,Dbt,Dbt)**

  is now: **DbSecondaryKeyCreate.secondaryKeyCreate(Db,Dbt,Dbt,Dbt)**

The 4.2 release of Berkeley DB requires at minimum a J2SE 1.3.1 certified Java virtual machine and associated classes to properly build and execute. To determine what version virtual machine you are running, enter:

```
java -version
```

at a command line and look for the version number. If you need to deploy to a version 1.1 or 1.0 Java environment, it may be possible to do so by not including the classes in the com.sleepycat.bdb package in the Java build process (however, that workaround has not been tested by us).

A few inconsistent methods have been cleaned up (for example, Db.close now returns void; previously, it returned an int which was always zero). The synchronized attributed has been toggled on some methods -- this is an attempt to prevent multithreaded applications from calling close or similar methods concurrently from multiple threads.

The Berkeley DB API has up until now been consistent across all language APIs. Although consistency has is benefits, it made our Java API look strange to Java programmers. Many methods have been renamed in this release of the Java API to conform with Java naming conventions. Sometimes this renaming was simply "camel casing", sometimes it required rewording. The mapping file for these name changes is in `dist/camel.pm`. The Perl script we use to convert code to the new names is called `dist/camelize.pl`, and may help with updating Java applications written for earlier versions of Berkeley DB.

Berkeley DB has a number of places where as a C library it uses function pointers to move into custom code for the purpose of notification of some event. In Java the best parallel is the registration of some class which implements an interface. In this version of Berkeley DB we have made an effort to make those interfaces more uniform and predictable. Specifically, DbEnvFeedback is now DbEnvFeedbackHandler, DbErrcall is DbErrorHandler and DbFeedback is DbFeedbackHandler. In every case we have kept the older interfaces and the older registration methods so as to allow for backward compatibility in this release. Expect them to be removed in future releases.

As you upgrade to this release of Berkeley DB you will notice that we have added an entirely new layer inside the package com.sleepycat.bdb. This was formerly the Greybird project by Mark Hayes. Sleepycat Software and Mark worked together to incorporate his work. We have done this in hopes of reducing the learning curve when using Berkeley DB in a Java project. When you upgrade you should consider switching to this layer as over time the historical classes and the new bdb package classes will be more and more integrated providing a simple yet powerful interface from Java into the Berkeley DB library.

Berkeley DB's Java API is now generated with SWIG [http://www.swig.org]. The new Java API is significantly faster for many operations.

Some internal methods and constructors that were previously public have been hidden or removed.

Packages found under com.sleepycat are considered different APIs into the Berkeley DB system. These include the core db api (com.sleepycat.db), the collections style access layer (com.sleepycat.bdb) and the now relocated XA system (com.sleepycat.xa).

Release 4.2: Queue access method

We have discovered a problem where applications that specify Berkeley DB's encryption or data checksum features on Queue databases with extent files, the database data will not be

protected. This is obviously a security problem, and we encourage you to upgrade these applications to the 4.2 release as soon as possible.

The Queue databases must be dumped and reloaded in order to fix this problem. First build the Berkeley DB 4.2 release, then use your previous release to dump the database, and the 4.2 release to reload the database. For example:

```
db-4.1.25/db_dump -P password -k database | db-4.2.xx/db_load -P password new_database
```

Note this is **only** necessary for Queue access method databases, where extent files were configured along with either encryption or checksums.

Release 4.2: DB_CHKSUM_SHA1

The flag to enable checksumming of Berkeley DB databases pages was renamed from DB_CHKSUM_SHA1 to DB_CHKSUM, as Berkeley DB uses an internal function to generate hash values for unencrypted database pages, not the SHA1 Secure Hash Algorithm. Berkeley DB continues to use the SHA1 Secure Hash Algorithm to generate hashes for encrypted database pages. Applications using the DB_CHKSUM_SHA1 flag should change that use to DB_CHKSUM; no other change is required.

Release 4.2: DB_CLIENT

The flag to create a client to connect to a RPC server was renamed from DB_CLIENT to DB_RPCCLIENT, in order to avoid confusion between RPC clients and replication clients. Applications using the DB_CLIENT flag should change that use to DB_RPCCLIENT; no other change is required.

Release 4.2: DB->del

In previous releases, the C++ Db::del and Java `Db.delete()` methods threw exceptions encapsulating the DB_KEYEMPTY (page 230) error in some cases when called on Queue and Recno databases. Unfortunately, this was undocumented behavior.

For consistency with the other Berkeley DB methods that handle DB_KEYEMPTY (page 230), this is no longer the case. Applications calling the Db::del and Java `Db.delete()` methods on Queue or Recno databases, and handling the DB_KEYEMPTY (page 230) exception specially, should be modified to check for a return value of DB_KEYEMPTY (page 230) instead.

Release 4.2: DB->set_cache_priority

In previous releases, applications set the priority of a database's pages in the Berkeley DB buffer cache with the DB->set_cache_priority method. This method is no longer available. Applications wanting to set database page priorities in the buffer cache should use the mempset_priority() method instead. The new call takes the same arguments and behaves identically to the old call, except that a DB_MPOOLFILE buffer cache file handle is used instead of the DB database handle.

Release 4.2: DB->verify

In previous releases, applications calling the DB->verify() method had to explicitly discard the DB handle by calling the DB->close() method. Further, using the DB handle in other ways after calling the DB->verify() method was not prohibited by the documentation, although such use was likely to lead to problems.

For consistency with other Berkeley DB methods, DB->verify() method has been documented in the current release as a DB handle destructor. Applications using the DB handle in any way (including calling the DB->close() method) after calling DB->verify() should be updated to make no further use of any kind of the DB handle after DB->verify() returns.

Release 4.2: DB_LOCK_NOTGRANTED

In previous releases, configuring lock or transaction timeout values or calling the DB_ENV->txn_begin() method with the DB_TXN_NOWAIT flag caused database operation methods to return DB_LOCK_NOTGRANTED (page 230), or throw a DbLockNotGrantedException exception. This required applications to unnecessarily handle multiple errors or exception types.

In the Berkeley DB 4.2 release, with one exception, database operations will no longer return DB_LOCK_NOTGRANTED (page 230) or throw a DbLockNotGrantedException exception. Instead, database operations will return DB_LOCK_DEADLOCK (page 230) or throw a DbDeadlockException exception. This change should require no application changes, as applications must already be dealing with the possible DB_LOCK_DEADLOCK (page 230) error return or DbDeadlockException exception from database operations.

The one exception to this rule is the DB->get() method using the DB_CONSUME_WAIT flag to consume records from a Queue. If lock or transaction timeouts are set, this method and flag combination may return DB_LOCK_NOTGRANTED (page 230) or throw a DbLockNotGrantedException exception.

Applications wanting to distinguish between true deadlocks and timeouts can configure database operation methods to return DB_LOCK_NOTGRANTED (page 230) or throw a DbLockNotGrantedException exception using the DB_TIME_NOTGRANTED flag.

The DB_ENV->lock_get() and DB_ENV->lock_vec() methods will continue to return DB_LOCK_NOTGRANTED (page 230), or throw a DbLockNotGrantedException exception as they have previously done.

Release 4.2: Replication

Replication initialization

In the Berkeley DB 4.2 release, replication environments must be specifically initialized by any process that will ever do anything other than open databases in read-only mode (that is, any process which might call any of the Berkeley DB replication interfaces or modify databases). This initialization is done when the replication database environment handle is opened, by specifying the DB_INIT_REP flag to the DB_ENV->open() method.

Database methods and replication clients

All of the DB object methods may now return DB_REP_HANDLE_DEAD when a replication client changes masters. When this happens the DB handle is no longer able to be used and the application must close the handle using the DB->close() method and open a new handle. This new return value is returned when a client unrolls a transaction in order to synchronize with the new master. Otherwise, if the application was permitted to use the original handle, it's possible the handle might attempt to access nonexistent resources.

DB_ENV->rep_process_message()

The DB_ENV->rep_process_message() method has new return values and an log sequence number (LSN) associated with those return values. The new argument is **ret_lsnp**, which is the returned LSN when the DB_ENV->rep_process_message() method returns DB_REP_ISPERM or DB_REP_NOTPERM. See Transactional guarantees (page 206) for more information.

Release 4.2: Client replication environments

In previous Berkeley DB releases, replication clients always behaved as if DB_TXN_NOSYNC behavior was configured, that is, clients would not write or synchronously flush their log when receiving a transaction commit or prepare message. However, applications needing a high level of transactional guarantee may need a write and synchronous flush on the client. By default in the Berkeley DB 4.2 release, client database environments write and synchronously flush their logs when receiving a transaction commit or prepare message. Applications not needing such a high level of transactional guarantee should use the environment's DB_TXN_NOSYNC flag to configure their client database environments to not do the write or flush on transaction commit, as this will increase their performance. Regardless of the setting of the DB_TXN_NOSYNC flag, clients will always write and flush on transaction prepare.

Release 4.2: Tcl API

The Tcl API included in the Berkeley DB 4.2 release requires Tcl release 8.4 or later.

Release 4.2: Upgrade Requirements

The log file format changed in the Berkeley DB 4.2 release. No database formats changed in the Berkeley DB 4.2 release.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.2.52 Change Log

Database or Log File On-Disk Format Changes:

1. Queue databases that use encryption or data checksum features with extent files will need to be dumped and reloaded prior to using with release 4.2. For more details please see the

"Upgrading Berkeley DB Applications, Queue Access Method" in the Berkeley DB Reference Guide included in your download package. [#8671]

2. The on-disk log format changed.

New Features:

1. Add support for a reduced memory footprint build of the Berkeley DB library. [#1967]

2. Add the DB_MPOOLFILE->set_flags interface which disallows the creation of backing filesystem files for in-memory databases. [#4224]

3. Add cache interfaces to limit the number of buffers written sequentially to allow applications to bound the time they will monopolize the disk. [#4935]

4. Support auto-deletion of log files. [#0040] [#6252]

5. The new Java DBX API for Berkeley DB allows Java programmers to use a familiar Java Collections style API, including Map, while interacting with the transactional Berkeley DB core engine. [#6260]

6. Support auto-commit with the DB->get method's consume operations. [#6954]

7. Add "get" methods to retrieve most settings. [#7061]

8. Add Javadoc documentation to the Berkeley DB release. [#7110]

9. Add support to Concurrent Data Store to allow duplication of write cursors. [#7167]

10. Add C++ utility classes for iterating over multiple key and data items returned from a cursor when using the DB_MULTIPLE or DB_MULTIPLE_KEY flags. [#7351]

11. Add CamelCased methods to the Java API. [#7396]

12. Add the DB_MPOOLFILE->set_maxsize interface to enforce a maximum database size. [#7582]

13. Add a toString() method for all Java *Stat classes (DbBtreeStat, DbHashStat, DbMpoolStat, etc.). This method creates a listing of values of all of the class member variables. [#7712]

Database Environment Changes:

1. Add cache interfaces to limit the number of buffers written sequentially to allow applications to bound the time they will monopolize the disk. [#4935]

2. Fix a bug which could cause database environment open to hang, in database environments supporting cryptography. [#6621]

3. Fix a bug where a database environment panic might result from an out-of-disk-space error while rolling back a page allocation. [#6694]

4. Fix a bug where a database page write failure, in a database environment configured for encryption or byte-swapping, could cause page corruption. [#6791]

5. Fix a bug where DB->truncate could drop core if there were active cursors in the database. [#6846]

6. Fix a bug where for databases sharing a physical file required a file descriptor per database. [#6981]

7. Fix a bug where the panic callback routine was only being called in the first thread of control to detect the error when returning DB_RUNRECOVERY. [#7019]

8. Fix a bug where a transaction which contained a remove of a subdatabase and an allocation to another subdatabase in the same file might not properly be aborted. [#7356]

9. Fix a bug to now disallow DB_TRUNCATE on opens in locking environments, since we cannot prevent race conditions. In the absence of locking and transactions, DB_TRUNCATE will truncate ANY file for which the user has appropriate permissions. [#7345]

10. Fix several bugs around concurrent creation of databases. [#7363]

11. Change methods in DbEnv that provide access to statistics information so that they now return instances of the proper classes. [#7395]

12. Replace the DB->set_cache_priority API with the DB_MPOOLFILE->set_priority API. [#7545]

13. Fix a bug where a failure during a creation of a subdatabase could then fail in the dbremove cleanup, causing a crash. [#7579]

14. Allow creating into a file that was renamed within the same transaction. [#7581]

15. Fix a bug where DB_ENV->txn_stat could drop core if there are more-than-expected active transactions. [#7638]

16. Change Berkeley DB to ignore user-specified byte orders when creating a database in an already existing physical file. [#7640]

17. Fix a bug where a database rename that is aborted would leak some memory. [#7789]

18. Fix a bug where files could not be renamed or removed if they were not writable. [#7819]

19. Fix a bug where an error during a database open may leak memory in the mpool region. [#7834]

20. Fix a bug where the DB_ENV->trickle_sync method could flush all of the dirty buffers in the cache rather than a subset. [#7863]

21. Fix a bug where an attempt to rename or remove an open file in the same transaction could succeed, even though this is not allowed and will not work on Windows. [#7917]

22. Fix a bug where if a recovery interval in the log contained only database opens then a recovery might report "Improper file close". [#7886]

23. Add a flag, DB_INIT_REP to DB_ENV->open to initialize Replication subsystem. [#8299]

24. Fix a bug where file remove and rename operations would not block each other if they were in different transactions. [#8340]

25. Change Berkeley DB to not propagate error returns from the application's rep_send function out of the Berkeley DB API. [#8496] [#8522]

26. Remove restriction that DB_TRUNCATE is not allowed on files with subdatabases. This restriction was introduced in 4.1.25. [#8852]

Concurrent Data Store Changes:

1. Fix a bug where opens with other threads/processes actively acquiring locks on database handles could deadlock. [#6286]

2. Add support to Concurrent Data Store to allow duplication of write cursors. [#7167]

General Access Method Changes:

1. Fix a bug where the truncate of a database with associated secondary databases did not truncate the secondaries. [#6585]

2. Fix a bug in which an out-of-disk condition during a transactional database create, remove, or rename could cause a crash. [#6695]

3. Fix a bug where system errors unknown to the C library could cause Berkeley DB utilities to drop core on Solaris. [#6728]

4. Fix a bug where Berkeley DB could overwrite incorrectly formatted files rather than returning an error to the application during open. [#6769]

5. Fix a bug DB handle reference counts were incorrect, leading to spurious warning about open DB handles. [#6818]

6. Fix a bug where cursor adjustments across multiple DB handles could fail. [#6820]

7. Fix a bug where a failure during open could result in a hang. [#6902]

8. Fix a bug where repeated failures during certain stages of opens could cause error messages to appear during recovery. [#7008]

9. Fix a bug in secondary indices with multiple threads calling DBC->put that resulted in DB_NOTFOUND being returned. [#7124]

10. Fix a bug where database verification might reference memory which was previously freed after reporting an error. [#7137]

11. Rename the DB_CHKSUM_SHA1 to DB_CHKSUM as Berkeley DB only uses SHA1 for encrypted pages, not for clear text pages. [#7095]

12. Fix a bug where DB->rename could fail silently if the underlying system rename call failed. [#7322]

13. Fix a bug where Berkeley DB failed to open a file with FCNTL locking and 0-length files. [#7345]

14. Prohibit the use of the DB_RMW flag on get operations for DB handles opened in transactional mode. [#7407]

15. Standardize when Berkeley DB will return DB_LOCK_NOTGRANTED, or throw DbLockNotGrantedException, versus returning DB_LOCK_DEADLOCK or throwing DbDeadlockException. Fix bugs in the C++ and Java APIs where DbException was thrown, encapsulating DB_LOCK_NOTGRANTED, rather than throwing DbLockNotGrantedException. [#7549]

16. Fix a bug where Berkeley DB could hang on a race condition if a checkpoint was running at the same time another thread was closing a database for the last time. [#7604]

17. Fix several bugs that made multiple filesystem level operations inside a single transaction break. [#7728]

18. Fix a memory leak in the abort path of a sub-database create. [#7790]

19. Fix a race condition with file close that could cause NULL pointer deference under load. [#8235]

20. Fix a bug to correct the calculation of the amount of space needed to return off page duplicates using the DB_MULTIPLE interface. [#8437]

21. Fix a bug where the duplicate data item count could be incorrect if a cursor was used to first overwrite and then delete a duplicate which was part of a set of duplicates large enough to have been stored outside the standard access method pages. [#8445]

22. Fix a bug where The DB_MULTIPLE interface might fail to return the proper duplicates in some edge cases. [#8485]

23. Fix a bug where DB->get(...DB_MULTIPLE) would not return a reasonable estimate of the buffer size required to return a set of duplicates. [#8513]

24. Fix a bug where the DbCursor.count method could return the wrong count in the case of small (on-page) duplicate sets, where a still-open cursor has been used to delete one of the duplicate data items. [#8851]

25. Fix a bug where a non-transactional cursor using DB_MULTIPLE_KEY could briefly be left pointing at an unlocked page. This could lead to a race condition with another thread deleting records resulting in the wrong record being deleted. [#8926]

26. Fix a bug where a key/data item could be lost if a cursor is used to do a delete, and then immediately used to do an insert which causes a set of duplicates to be shifted to an off-page Btree. [#9085]

Btree Access Method Changes:

1. Fix a bug where a deleted item could be left on a database page causing database verification to fail. [#6059]
2. Fix a bug where a page may be left pinned in the cache if a deadlock occurs during a DB->put operation. [#6875]
3. Fix a bug where a deleted record may not be removed from a Btree page if the page is split while another cursor is trying to delete a record on the page. [#6059]
4. Fix a bug where records marked for deletion were incorrectly counted when retrieving in a Btree by record number. [#7133]
5. Fix a bug where a page and lock were left pinned if an application requested a record number past the end of the file when retrieving in a Btree by record number. [#7133]
6. Fix a bug where deleted keys were included in the key count for the DB->stat call. [#7133]
7. Fix a bug where specifying MULTIPLE_KEY and NEXT_DUP to the bulk get interfaces might return the wrong data if all the duplicates could not fit in a single buffer. [#7192]
8. Remove assertions that triggered failures that were correct executions. [#8032]
9. Fix a bug where duplicate data items were moved onto overflow pages before it was necessary. [#8082]
10. Fix a bug where the DB->verify method might incorrectly complain about a tree's overflow page reference count. [#8061]
11. Fix a bug that could cause DB_MULTIPLE on a Btree database to return an incorrect data field at the end of buffer. [#8442]
12. Fix a bug where DBC->c_count was returning an incorrect count if the cursor was positioned on an item that had just been deleted. [#8851]
13. Remove the test for bt_maxkey in the Btree put code. If it is set to 1 it can cause an infinite loop. [#8904]

Hash Access Method Changes:

1. Fix a bug where Hash databases could be corrupted on filesystems that do not zero-fill implicitly created blocks. [#6588]
2. Fix a bug where creating a Hash database with an initial size larger than 4GB would fail. [#6805]
3. Fix a bug where a page in an unused hash bucket might not be empty if there was a disk error while writing the log record for the bucket split. [#7035]

4. Fix a bug where two threads opening a hash database at the same time might deadlock. [#7159]

5. Fix a bug where a hash cursor was not updated properly when doing a put with DB_NODUPDATA specified. [#7361]

6. Fix a bug that could cause DB_MULTIPLE_KEY on Hash databases to return improper results when moving from a key with duplicates to a key without duplicates. [#8442]

Queue Access Method Changes:

1. Fix a bug where opening an in-memory Queue database with extent size specified will dump core. [#6795]

2. Support auto-commit with the DB->get method's consume operations. [#6954]

3. Fix a bug where calling the sync method on a queue database with extents may hang if there are active consumers. [#7022]

4. Fix a bug where a get(...MULTIPLE...) might lead to an infinite loop or return the wrong record number(s) if there was a deleted record at the beginning of a page or the buffer was filled exactly at the end of a page. [#7064]

5. Fix a bug where a database environment checkpoint might hang if a thread was blocked waiting for a record while doing a DB_CONSUME_WAIT on a Queue database. [#7086]

6. Fix a bug where queue extent files would not be removed if a queue with extents was removed and its record numbers wrapped around the maximum record number. [#7191]

7. Fix a bug where a DB->remove of an extent based Queue with a small number of pages per extent would generate a segmentation fault. [#7249]

8. Fix a bug where verify and salvage on queues with extent files did not consider the extent files. [#7294]

9. Fix a bug when transaction timeouts are set in the environment they would get applied to some non-transactional operations and could cause a failure during the abort of a queue operation. [#7641]

10. Fix a bug when the record numbers in a queue database wrap around at 232, a cursor positioned on a record near the head of the queue that is then deleted, may return DB_NOTFOUND when get is specified with DB_NEXT rather than the next non-deleted record. [#7979]

11. Fix a bug where a record lock will not be removed when the first record in the queue is deleted without a transaction (not using DB_CONSUME). [#8434]

12. Fix a bug where byte swapping was not handled properly in queue extent files. [#8358]

13. Fix a bug where Queue extent file pages were not properly typed, causing the extent files not to use encryption or checksums, even if those options had been specified. This fix requires a database upgrade for any affected Queue databases. [#8671]

14. Fix a bug where truncating a queue with extents may fail to remove the last extent file. [#8716]

15. Fix a bug where a rename or remove of a QUEUE database with extents might leave empty extent files behind. [#8729]

16. Fix a bug where on Windows operating systems a "Permission denied" error may be raised if a Queue extent is reopened while it is in the process of being unlinked. [#8710]

Recno Access Method Changes:

1. Fix a bug where the DB->truncate method may return the wrong record count if there are deleted records in the database. [#6788]

2. Fix a bug where internal nodes of Recno trees could get wrong record count if a log write failed and the log was later applied during recovery. [#6841]

3. Fix a bug where a cursor next operation could infinitely loop after deleting a record, when the deleted record was immediately followed by implicitly created records. [#8133]

C++-specific API Changes:

1. Document the DB->del method can return DB_KEYEMPTY for Queue or Recno databases. The C++ and Java APIs now return this value rather than throwing an exception. [#7030]

2. Add "get" methods to retrieve most settings. [#7061]

3. Fix a bug where applications calling DB->verify from the C++ or Java APIs could drop core. Change the DB->verify method API to act as a DB handle destructor. [#7418]

4. Add utility classes for iterating over multiple key and data items returned from a cursor when using the DB_MULTIPLE or DB_MULTIPLE_KEY flags. These classes, DbMultipleDataIterator, DbMultipleKeyDataIterator, and DbMultipleRecnoDataIterator, mirror the DB Java API and are provided as replacements for the C macros, DB_MULTIPLE_INIT, DB_MULTIPLE_NEXT, DB_MULTIPLE_KEY_NEXT, and DB_MULTIPLE. [#7351]

5. Fix a bug DbException was thrown, encapsulating DB_LOCK_NOTGRANTED, rather than throwing DbLockNotGrantedException. [#7549]

6. Add the DbEnv handle to exceptions thrown by the C++ and Java APIs, where possible. [#7303]

7. Fix a bug in the C++ DbEnv::set_rep_transport signature so that the envid parameter is signed. [#8303]

8. Make the fields of DB_LSN public in the DbLsn class. [#8422]

Java-specific API Changes:

a. Db.put(), Dbc.get() and Dbc.put() preserve key size

b. Dbc.get() returns DB_KEYEMPTY rather than throwing an exception

c. The return type of Db.close() is now void. [#7002]

1. New Java API (com.sleepycat.dbx.*) for the transactional storage of data using the Java Collections design pattern. [#6569]

2. Fix a bug in the Java Dbt.get_recno_key_data() method when used inside callbacks. [#6668]

3. Fix Java DbMpoolStat class to match the DB_MPOOL_STAT struct. [#6821]

4. Fix a bug where Dbc.put expected key data even if the key was unused. [#6932]

5. Fix a bug in the Java API secondary_key_create callback where memory was freed incorrectly, causing JVM crashes. [#6970]

6. Re-implement the Java API to improve performance and maintenance. Fix several inconsistencies in the Java API:

7. Document the DB->del method can return DB_KEYEMPTY for Queue or Recno databases. The C++ and Java APIs now return this value rather than throwing an exception. [#7030]

8. Add "get" methods to retrieve most settings. [#7061]

9. Add Javadoc documentation to the Berkeley DB release. [#7110]

10. Fix a bug that caused potential memory corruption when using the Java API and specifying the DB_DBT_REALLOC flag. [#7215]

11. Add the DbEnv handle to exceptions thrown by the C++ and Java APIs, where possible. [#7303]

12. Map existing c-style API to a more Java camel case API with Java style naming. Retained deprecated older API for the 4.2 release for backwards support in all cases except callback interfaces. Also overloaded methods such as get/pget() into multiple different get() calls to clean up call structure. [#7378]

13. Add CamelCased methods to the Java API. [#7396]

14. Fix a bug where applications calling DB->verify from the C++ or Java APIs could drop core. Change the DB->verify method API to act as a DB handle destructor. [#7418]

15. Fix a bug DbException was thrown, encapsulating DB_LOCK_NOTGRANTED, rather than throwing DbLockNotGrantedException. [#7549]

16. Add a toString() method for all Java *Stat classes (DbBtreeStat, DbHashStat, DbMpoolStat, etc.). This method creates a listing of values of all of the class member variables. [#7712]

17. Remove Db.fd() method from Java API as it has no value to a Java programmer. [#7716]

18. Add an accessible timeout field in the DbLockRequest class, needed for the DB_LOCK_GET_TIMEOUT operation of DbEnv.lockVector. [#8043]

19. Fix replication method calls from Java API. [#8467]

20. Fix a bug where exception returns were inconsistent. [#8622]

21. Change the Java API so that it throws an IllegalArgumentException rather than a DbException with the platform-specific EINVAL. [#8978]

Tcl-specific API Changes:

1. Add "get" methods to retrieve most settings. [#7061]

2. Brought Tcl's $env set_flags command up to date with available flags. [#7385]

3. Update Berkeley DB to compile cleanly against the Tcl/Tk 8.4 release. [#7612]

4. Made txn_checkpoint publicly available. [#8594]

RPC-specific Client/Server Changes:

1. Fix two bugs in the RPC server where incorrect handling of illegal environment home directories caused server crashes. [#7075]

2. Fix a bug where the DB_ENV->close method would fail in RPC clients if the DB_ENV->open method was never called. [#8200]

Replication Changes:

1. Write prepare records synchronously on replication clients so that prepare operations are always honored in the case of failure. [#6416]

2. Change replication elections so that the client with the biggest LSN wins, and priority is a secondary factor. [#6568]

3. Fix a bug where replicas could not remove log files because the checkpoint lsn was not being updated properly. [#6620]

4. Force prepare records out to disk regardless of the setting of the DB_TXN_NOSYNC flag. [#6614]

5. Add a new flag, DB_REP_NOBUFFER, which gets passed to the rep_send function specified in DBENV->rep_set_transport, to indicate that the message should not be buffered on the master, but should be immediately transmitted to the client(s). [#6680]

6. Fix a replication election bug where Berkeley DB could fail to elect a master even if a master already existed. [#6702]

7. Allow environment wide setting of DB_AUTO_COMMIT on replication clients. [#6732]

8. Fix a replication bug where a client coming up in the midst of an election might not participate in the election. [#6826]

9. Add log_flushes when sites become replication masters. If log_flush fails, panic the environment since the clients already have the commits. [#6873]

10. Fix a replication bug where a brand new client syncing up could generate an error on the master. [#6927]

11. Fix a bug where clients synchronize with the master when they come up with the same master after a client-side disconnect or failures. [#6986]

12. Fix several bugs in replication elections turned up by test rep005. [#6990]

13. Fix a bug where aborted hash group allocations were not properly applied on replicas. [#7039]

14. Fix race conditions between running client recovery and other threads calling replication and other Berkeley DB functions. [#7402] [#8035]

15. Use shared memory region for all replication flags. [#7573]

16. Fix a bug where log archive on clients could prematurely remove log files. [#7659]

17. Return an error if a non-replication dbenv handle attempts to write log records to a replication environment. [#7752]

18. Fix a race condition when clients applied log records, where we would store a log record locally and then never notice we have it, and need to re-request it from the master, causing the client to get far behind the master. [#7765]

19. Fix inconsistencies between the documentation and actual code regarding when replication methods can be called. [#7775]

20. Fix a bug where Berkeley DB would wait forever if a NEWMASTER message got dropped. [#7897]

21. Fix a bug where the master environment ID did not get set when you called DBENV->rep_start as a master. [#7899]

22. Fix a bug where operations on a queue database will not get replicated if the transactions that include the operations are committed out of order with the operations. [#7904]

23. Fix bugs in log_c_get where an invalid LSN could access invalid addresses. Fix bug in elections where a client upgrading to master didn't write a txn_recycle record. [#7964]

24. Fix a bug where REP_VERIFY_FAIL during client recovery wasn't being handled. [#8040]

25. Return an error if the application calls rep_process_message before calling rep_start when starting. [#8057]

26. Fix a bug to ensure that replication generation numbers always increase and are never reset to 1. [#8136]

27. Modify log message retransmission protocol to efficiently handle the case where a large number of contiguous messages were dropped at once. [#8182] [#8169] [#8188]

28. Fix a bug where using the wrong mutex in replication which under certain conditions could cause replication to hang. Also fix a bug where incorrectly setting the checkpoint LSN could cause recovery to take a very long time. [#8183]

29. Fix bug where a message could get sent to an invalid master. [#8184]

30. Fix a bug where a local variable in log_archive was not initialized. [#8230]

31. Fix a bug where elections could hang. [#8254]

32. Fix a bug to ensure that we can always remove/re-create the temporary replication database after a failure. [#8266]

33. Add a flag, DB_INIT_REP to DB_ENV->open to initialize Replication subsystem. [#8299]

34. Add new ret_lsnp argument to rep_process_message so that LSNs can be returned to clients on permanent records. Add new lsnp arg to the send callback function so that the master can know the LSNs of records as well. [#8308]

35. Narrow the window where we block due to client recovery. [#8316]

36. Fix a bug in log_c_incursor where we would not detect that a record was already in the buffer. [#8330]

37. Fix a bug that would allow elections to be managed incorrectly. [#8360]

38. Fix a bug where replicas were not maintaining meta->last_pgno correctly. [#8378]

39. Fix a bug in truncating log after recovery to a timestamp or replication-based recovery. [#8387]

40. Fix a bug where a checkpoint record written as the first record in a log could cause recovery to fail. [#8391]

41. Fix a bug where a client would return DB_NOTFOUND instead of DB_REP_OUTDATED when it was unable to synchronize with the master because it ran out of log records. [#8399]

42. Fix a bug where log file changes were not handled properly in replication. [#8400] [#8420]

43. Fix a bug where checking for invalid log header data could fail incorrectly. [#8460]

44. Fix a bug where DB_REP_PERMANENT was not being set when log records were re-transmitted. [#8473]

45. Modify elections so that all participants elect in the same election generation. [#8590]

46. Fix bug where rep_apply was masking an error return. Also return DB_RUNRECOVERY if the replication client cannot commit or checkpoint. [#8636]

47. Fix a bug to update the last_pgno on the meta page on free as well as alloc. [#8637]

48. Fix a bug to roll back the LSN on a queue database metapage if we're going to truncate the log. Fix a bug in MASTER_CHECK so we don't apply log messages from an unknown master. Fix a bug to perform a sync on rep_close. [#8601]

49. Fix a bug so that we reset the LSN when putting pages on the free list. [#8685]

50. Fix a bug where replication was not properly calling db_shalloc. [#8811]

51. Fix a bug where replication flags were getting set in multiple steps which could cause an Assertion Failure in log_compare. [#8889]

52. Fix a bug where open database handles could cause problems on clients. [#8936]

53. Fix a bug where in dbreg code where an fnp with an invalid fileid could be found on the lp->fq list. [#8963]

54. Fix a bug where a reader on a replication client could see partial updates when replicating databases with off page duplicates or overflow records. [#9041]

55. Fix a bug that could result in a self deadlock in dbreg under replication. [#9138]

56. Fix a memory leak in replication. [#9255]

XA Resource Manager Changes:

1. Fix a bug where a failed write during XA transaction prepare could result in a checksum error in the log. [#6760]

2. Fix a bug where we were not properly handling DB_AUTO_COMMIT in XA transactions and where we were not honoring the XA transaction during an XA-protected open. [#6851]

3. Add infrastructure support for multithreaded XA. [#6865]

4. Display XA status and ID as part of db_stat -t output. [#6413]

Locking Subsystem Changes:

a. failure to remove dirty read locks prior to aborting a transaction,

b. calling upgrade on other than WWRITE locks,

c. failure to remove expired locks from the locker queue,

d. clearing the lock timeout before looking at it. [#7267]

1. Fix a bug where locks were not cleared in an off-page duplicate cursor. [#6950]

2. Fix a bug where a deadlock may not be detected if dirty reads are enabled and the deadlock involves an aborting transaction. [#7143]

3. Fix a bug where a transaction doing updates while using dirty read locking might fail while aborting the transaction with a deadlock. Several other locking issues were also fixed:

4. Fix a bug when dirty reads are enabled a writer might be blocked on a lock that it had previously obtained. Dirty readers would also wait behind regular readers when they could have safely read a page. [#7502]

5. Fix a bug where a DB->put using CDB gets a lock timeout then the error "Closing already closed cursor". [#7597]

6. Modify the maximum test-and-set mutex sleep for logical page locks at 10ms, everything else at 25ms. [#7675]

7. Fix a bug where the DB_LOCK_TIMEOUT mode of env->lock_vec could hang. [#7682]

8. Fix a bug where running with only transaction timeouts for deadlock detection might deadlock without being detected if more than one transaction times out while trying to avoid searching a Btree on repeated inserts. [#7787]

9. Fix a bug that could cause detection to not run when there was a lock that should be timed out. [#8588]

10. Fix a bug with using dirty reads with subtransactions. If a writing subtransaction aborts and then is blocked, the deadlock may not be detected. [#9193]

11. Fix a bug where handle locks were not being correctly updated when releasing read locks during transaction prepare. [#9275]

Logging Subsystem Changes:

1. Fix a bug where if a write error occurred while committing a transaction with DB_WRITE_NOSYNC enabled the transaction may appear to be committed in the log while it was really aborted. [#7034]

2. Fix a bug where multiprocess applications could violate write-ahead logging requirements if one process wrote a log record but didn't flush it, the current log file then changed, and another process wrote a database page before the log record was written to disk. [#6999]

3. Fix a bug where fatal recovery could fail with a "Transaction already committed" error if recovery had been run and there are no active transactions in the part of the log following the last checkpoint. [#7234]

4. Fix a bug where recovery would fail to put freed pages onto the free list, when both committed and aborted subtransactions that allocated new pages were present. This only affected prepared transactions. [#7403]

5. Fix a bug where open errors during recovery get propagated unless they are reporting missing files, which might correctly have been removed. [#7578]

6. Fix a bug so that we now validate a log file before writing to it. [#7580]

7. Fix a bug where Berkeley DB could display the unnecessary error message "DB_LOGC->get: short read" during recovery. [#7700]

8. Fix a bug where recovery may fail if it tries to reallocate a page to a file that is out of space. [#7780]

9. Change Berkeley DB so that operations on databases opened in a non-transactional mode do not write records into the database logs. [#7843]

10. Fix a bug where Berkeley DB could timeout waiting for locks (on Queue databases) during recovery. [#7927]

11. Fix a bug in truncating log after recovery to a timestamp or replication-based recovery. [#8387]

12. Fix a bug where recovery can be slow if the log contains many opens of files which contain multiple databases. [#8423]

13. Fix a bug where a file id could be used before its open was logged. [#8496]

14. Fix a bug where recovery would partially undo a database create if the transaction which created it spanned log files and not all of the log files were present during recovery. [#9039]

Memory Pool Subsystem Changes:

1. Fix a bug where checksummed files could not be read on different endian systems. [#6429]

2. Fix a bug where read-only databases were not mapped into memory but were instead read through the Berkeley DB buffer cache. [#6671]

3. Fix a bug where Berkeley DB could loop infinitely if the cache was sized so small that all of its pages were simultaneously pinned by the application. [#6681]

4. Fix a bug where DbEnv.sync could fail to write a page if another thread unpinned the page at the same time and there were no other pages in that hash bucket. [#6793]

5. Fix a bug where threads of control may hang if multiple threads of control are opening and closing a database at the same time. [#6953]

6. Fix a bug where a database created without checksums but later opened with checksums would result in a checksum error. [#6959]

7. Fix a bug where a multiprocess application suite could see incorrect data if one process opened a non-checksummed database

8. Change to avoid database open and flush when handles are discarded, if the handle was never used to write anything. [#7232]

9. Fix a bug where applications dirtying the entire cache in a single database operation would see large performance degradation. [#7273]

10. Fix a bug where contention in the buffer pool could cause the buffer allocation algorithm to unnecessarily sleep waiting for buffers to be freed. [#7572]

Transaction Subsystem Changes:

1. Fix a bug where disk write errors in encrypted database environments, causing transaction abort, could corrupt the log. [#6768]

2. Fix a bug where catastrophic recovery may fail on a log which has a prepared transaction which aborted the allocation of a new page and was rolled forward previously by another recovery session. [#6790]

3. Fix a bug where a transaction that contains a database truncate followed by page allocations, may not properly undo the truncate if aborted. [#6862]

4. Fix a bug which causes Berkeley DB to checkpoint quiescent database environments. [#6933]

5. Fix a bug where if a transaction prepare fails while writing the prepare log record, and it contains a subtransaction which did an allocation later, recovery of the database may fail with a log sequence error. [#6874]

6. Do not abort prepared but not yet completed transactions when closing an environment. [#6993]

7. Fix a bug where operations on the source of a rename in the same transaction would fail. [#7537]

8. Fix a bug where a parent transaction which aborts when it tries to write its commit record could fail with a log sequence error, if the parent transaction has an aborted child transaction which allocated a new page from the operating system. [#7251]

9. Fix a bug where Berkeley DB could try to abort a partial transaction because it contained a partial subtransaction. [#7922]

10. Fix a bug where Berkeley DB could drop core when transactions were configured without locking support. [#9255]

Utility Changes:

1. Fix a bug where db_load could core dump or corrupt record numbers by walking off the end of a string. [#6985]

2. Fix a bug where db_load could run out of locks when loading large numbers of records. [#7173]

3. Fix a bug where db_dump could drop core when salvaging unaligned entries on a Btree page. [#7247]

4. Fix a bug where hash statistics did not include overflow items in the count of database data elements. [#7473]

5. Fix a bug where an corruption in an overflow page list could cause DB->verify to infinitely loop. [#7663]

6. Fix a bug where verify could display extraneous error messages when verifying a Btree with corrupt or missing pages. [#7750]

7. Fix a bug that could cause the db_stat utility to display values larger than 100 for various percentages. [#7779]

8. Fix a memory overflow bug in db_load. [#8124]

9. Fix a minor leak when verifying queue databases. [#8620]

Configuration, Documentation, Portability and Build Changes:

1. Add support for a reduced memory footprint build of the Berkeley DB library. [#1967]

2. Change DB_SYSTEM_MEM on Windows to fail immediately when opening an environment whose regions were deleted on last close. [#4882]

3. Update queue.h to current FreeBSD version. [#5494]

4. Support for and certification under Tornado 2.2/VxWorks 5.5. [#5522]

5. Add support for IBM OS/390 using the IBM C compiler. [#6486]

6. Specify -pthread as a compile flag for Tru64 systems, not just as a linker flag. [#6637]

7. Remove automatic aggregate initialization for non-ANSI compilers. [#6664]

8. Fix a link error ("GetLongPathNameA could not be located in the dynamic link library KERNEL32.dll") that prevented Berkeley DB from loading on Windows NT. [#6665]

9. Remove use of U suffix in crypto build to denote unsigned integers for non-ANSI compilers. [#6663]

10. Fix Java API documentation problems where API return values were int and should have been void, or vice versa. [#6675]

11. Add an include of <sys/fcntl.h> for old Solaris systems with the directio call. [#6707]

12. Fix Java API documentation problem where the Db.associate call was missing a DbTxn handle. [#6714]

13. Clean up source based on gcc's -Wmissing-prototypes option. [#6759]

14. Ignore pread/pwrite interfaces on NCR's System V R 4.3 system. [#6766]

15. Fix an interface compatibility with Sendmail and Postfix releases. [#6769]

16. Fix warnings when the Tcl API was built without TEST_CONFIG defined. [#6789]

17. Change Win32 mutexes to use the shared code for all mutexes to fix handle leak. [#6822] [#6853]

18. Fix the Windows/Tcl API export list for Berkeley DB XML. [#6931]

19. Add the --enable-mingw configuration option to build Berkeley DB for MinGW. [#6973]

20. Add a CPU pause to the mutex spinlock code to improve performance on newer Pentium CPUs. [#6975]

21. Upgrade read-only file descriptors to read-write during checkpoint, it's an error to call FlushFileBuffers on a read-only Windows file handle. [#7051]

22. Fix configure so that Java applications on HP/UX can access RPC environments. [#7066]

23. Update Berkeley DB to use libtool 1.5 to allow building of shared libraries on various platforms. This should not be visible except for changes to the Makefile and internal build procedures. [#7080]

24. Fix a bug where the configure script displayed incorrect default installation directory information. [#7081]

25. Fix a signed/unsigned warning with some Windows compilers. [#7100]

26. Fix macro redefinition conflicts between queue.h and Vc7\PlatformSDK\Include\WinNT.h when building with Visual Studio.NET 7.0. [#7103]

27. Add a loop to retry system calls that return EBUSY. Also limit retries on EINTR to 100 times. [#7118]

28. Fix a bug in our use of GetDiskFreeSpace that caused access violations on some versions of Windows with DB_DIRECT_DB. [#7122]

29. Fix a bug where regions in system memory on Windows were incorrectly reinitialized because the magic number was overwritten. [#7127]

30. Change version provided to Tcl's package system to reflect Berkeley DB's major and minor number. [#7174]

31. Support for the Berkeley DB Embedix port has been removed. [#7209]

32. Merge all public C++ headers into db_cxx.h, which fixes name clashes between Berkeley DB headers and system headers (specifically mutex.h). [#7221]

33. Fix a bug where the configured Makefile could try and build objects for which there were no existing rules. [#7227]

34. Port the ex_repquote example to Windows. [#7328]

35. Fix a race in the ARM/gcc mutex code which could cause almost anything bad you can imagine. [#7468]

36. Fix a bug where shared region removal could hang. [#7613]

37. Fix a bug so that when using Java in Debug mode on Windows, automatically pick the Debug DLL. [#7722]

38. Fix configure --disable-shared so that it now creates a Makefile that installs static libraries that look the same as a regular shared build. This flag will create a libdb<major>.<minor>.a and make a libdb.a that is a symlink to it. [#7755]

39. Add support for OS/390 2.10 and all versions of z/OS. [#7972]

40. Support Java builds on Windows with spaces in the project path. [#8141]

41. Fix a bug where Berkeley DB mutex locking code for OS X was not multiprocessor safe. [#8255]

42. Add an error to DB_ENV->set_flags if the OS does not support Direct

43. Enable verbose error logging from the test suite on Windows. [#8634]

44. Fix a bug with DLL linking on Cygwin under Windows. [#8628]

45. Add support for JDK on HP/UX. [#8813]

46. Fix a bug where pathnames longer than 2KB could cause processes to core dump. [#8886]

47. Fix a bug in VxWorks when yielding the CPU, so that we delay at least one tick. [#9061]

Chapter 40. Upgrading Berkeley DB 4.2 applications to Berkeley DB 4.3

Release 4.3: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.2 release interfaces to the Berkeley DB 4.3 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.3: Java

The Berkeley DB Java API has changed significantly in the 4.3 release, in ways incompatible with previous releases. This has been done to provide a consistent Java-like API for Berkeley DB as well as to make the Berkeley DB Java API match the API in Berkeley DB Java Edition, to ease application-porting between the two libraries.

Here is a summary of the major changes:

- Db -> Database
- Dbc -> Cursor
- Dbt -> DatabaseEntry
- DbEnv -> Environment
- DbTxn -> Transaction
- Db.cursor -> Database.openCursor
- Dbc.get(..., DbConstants.DB_CURRENT) -> Cursor.getCurrent(...)

1. The low-level wrapper around the C API has been moved into a package called com.sleepycat.db.internal.
2. There is a new public API in the package com.sleepycat.db.
3. All flags and error numbers have been eliminated from the public API. All configuration is done through method calls on configuration objects.
4. All classes and methods are named to Java standards, matching Berkeley DB Java Edition. For example:
5. The statistics classes have "getter" methods for all fields.
6. In transactional applications, the Java API infers whether to auto-commit operations: if an update is performed on a transactional database without supplying a transaction, it is implicitly auto-committed.

7. The com.sleepycat.bdb.* packages have been reorganized so that the binding classes can be used with the base API in the com.sleepycat.db package. The bind and collection classes are now essentially the same in Berkeley DB and Berkeley DB Java Edition. The former com.sleepycat.bdb.bind.* packages are now the com.sleepycat.bind.* packages. The former com.sleepycat.bdb, com.sleepycat.bdb.collections, and com.sleepycat.bdb.factory packages are now combined in the new com.sleepycat.collections package.

8. A layer of the former collections API has been removed to simplify the API and to remove the redundant implementation of secondary indices. The former DataStore, DataIndex, and ForeignKeyIndex classes have been removed. Instead of wrapping a Database in a DataStore or DataIndex, the Database object is now passed directly to the constructor of a StoredMap, StoredList, etc.

Release 4.3: DB_ENV->set_errcall, DB->set_errcall

The signature of the error callback passed to the DB_ENV->set_errcall() and DB->set_errcall() methods has changed in the 4.3 release. For example, if you previously had a function such as this:

```
void handle_db_error(const char *prefix, char *message);
```

it should be changed to this:

```
void handle_db_error(const DB_ENV *dbenv,
    const char *prefix, const char *message);
```

This change adds the DB_ENV handle to provide database environment context for the callback function, and incidentally makes it clear the message parameter cannot be changed by the callback.

Release 4.3: DBcursor->c_put

The 4.3 release disallows the DB_CURRENT flag to the DBC->put() method after the current item referenced by the cursor has been deleted. Applications using this sequence of operations should be changed to do the put without first deleting the item.

Release 4.3: DB->stat

The 4.3 release adds transactional support to the DB->stat() method.

Application writers can simply add a NULL **txnid** argument to the DB->stat() method calls in their application to leave the application's behavior unchanged.

Release 4.3: DB_ENV->set_verbose

The 4.3 release removes support for the DB_ENV->set_verbose() method flag DB_VERB_CHKPOINT. Application writers should simply remove any use of this flag from their applications.

The 4.3 release redirects output configured by the DB_ENV->set_verbose() method from the error output channels (see the DB_ENV->set_errfile() and DB_ENV->set_errcall() methods for more information) to the new DB_ENV->set_msgcall() and DB_ENV->set_msgfile() message output channels. This change means the error output channels are now only used for errors, and not for debugging and performance tuning messages as well as errors. Application writers using DB_ENV->set_verbose() should confirm that output is handled appropriately.

Release 4.3: Logging

In previous releases, the DB_ENV->set_flags() method flag DB_TXN_NOT_DURABLE specified that transactions for the entire database environment were not durable. However, it was not possible to set this flag in environments that were part of replication groups, and physical log files were still created. The 4.3 release adds support for true in-memory logging for both replication and non-replicated sites.

Existing applications setting the DB_TXN_NOT_DURABLE flag for database environments should be upgraded to set the DB_LOG_INMEMORY flag instead.

In previous releases, log buffer sizes were restricted to be less than or equal to the log file size; this restriction is no longer required.

Release 4.3: DB_FILEOPEN

The 4.3 release removes the DB_FILEOPEN error return. Any application check for the DB_FILEOPEN error should be removed.

Release 4.3: ENOMEM and DbMemoryException

In versions of Berkeley DB before 4.3, the error **ENOMEM** was used to indicate that the buffer in a DBT configured with DB_DBT_USERMEM was too small to hold a key or data item being retrieved. The 4.3 release adds a new error, `DB_BUFFER_SMALL`, that is returned in this case.

The reason for the change is that the use of **ENOMEM** was ambiguous: calls such as DB->get() or DBC->get() could return **ENOMEM** either if a DBT was too small or if some resource was exhausted.

The result is that starting with the 4.3 release, C applications should always treat **ENOMEM** as a fatal error. Code that checked for the **ENOMEM** return and allocated a new buffer should be changed to check for `DB_BUFFER_SMALL`.

In C++ applications configured for exceptions, a DbMemoryException will continue to be thrown in both cases, and applications should check the errno in the exception to determine which error occurred.

In Java applications, a **DbMemoryException** will be thrown when a **Dbt** is too small to hold a return value, and an **OutOfMemoryError** will be thrown in all cases of resource exhaustion.

Release 4.3: Replication

The 4.3 release removes support for logs-only replication clients. Use of the DB_REP_LOGSONLY flag to the DB_ENV->rep_start() should be replaced with the DB_REP_CLIENT flag.

The 4.3 release adds two new arguments to the DB_ENV->rep_elect() method, **nvotes** and **flags**. The **nvotes** argument sets the required number of replication group members that must participate in an election in order for a master to be declared. For backward compatibility, set the **nvotes** argument to 0. The flags argument is currently unused and should be set to 0. See DB_ENV->rep_elect() method or "Replication Elections" for more information.

In the 4.3 release it is no longer necessary to do a database environment hot backup to initialize a replication client. All that is needed now is for the client to join the replication group. Berkeley DB will perform an internal backup from the master to the client automatically and will run recovery on the client to bring it up to date with the master.

Release 4.3: Run-time configuration

The signatures of the db_env_set_func_ftruncate and db_env_set_func_seek functions have been simplified to take a byte offset in one parameter rather than a page size and a page number.

Release 4.3: Upgrade Requirements

The log file format changed in the Berkeley DB 4.3 release. No database formats changed in the Berkeley DB 4.3 release.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.3.29 Change Log

Database or Log File On-Disk Format Changes:

1. The on-disk log format has changed.

New Features:

1. Add support for light weight, transactionally protected Sequence Number generation. [#5739]

2. Add support for Degree 2 isolation. [#8689]

3. Add election generation information to replication to support Paxos compliance. [#9068]

4. Add support for 64-bit and ANSI C implementations of the RPCGEN utility. [#9548]

Database Environment Changes:

1. Fix a bug where the permissions on system shared memory segments did not match the mode specified in the DB_ENV->open() method. [#8921]

2. Add a new return error from the DB_ENV->open() method call, DB_VERSION_MISMATCH, which is returned in the case of an application compiled under one version of Berkeley DB attempting to open an environment created under a different version. [#9077]

3. Add support for importing databases from a transactional database environment into a different environment. [#9324]

4. Fix a bug where a core dump could occur if a zero-length database environment name was specified. [#9233]

5. Increase the number of environment regions to 100. [#9297]

6. Remove the DB_ENV->set_verbose() method flag DB_VERB_CHKPOINT. [#9405]

7. Fix bugs where database environment getters could return incorrect information after the database environment was opened, if a different thread of control changed the database environment values. Fix bugs where database environment getter/setter functions could race with other threads of control. [#9724]

8. Change the DbEnv.set_lk_detect method to match the DbEnv.open semantics. That is, DbEnv.set_lk_detect may be called after the database environment is opened, allowing applications to configure automatic deadlock detection if it has not yet been configured. [#9724]

9. Fix cursor locks for environments opened without DB_THREAD so that they use the same locker ID. This eliminates many common cases of application self-deadlock, particularly in CDS. [#9742]

10. Fix a bug in DB->get_env() in the C API where it could return an error when it should only return the DB_ENV handle. C++ and Java are unchanged. [#9828]

11. Fix a bug where we only need to initialize the cryptographic memory region when MPOOL, Log or Transactions have been configured. [#9872]

12. Change private database environments at process startup to only allocate the heap memory required at any particular time, rather than always allocating the maximum amount of heap memory configured for the environment. [#9889]

13. Add a method to create nonexistent intermediate directories when opening database files. [#9898]

14. Add support for in-memory logging within database environments. [#9927]

15. Change Berkeley DB so configuring a database environment for automatic log file removal affects all threads in the environment, not just the DbEnv handle in which the configuration call is made. [#9947]

16. Change the signature of the error callback passed to the DB_ENV->set_errcall and DB->set_errcall methods to add a DB_ENV handle, to provide database environment context for the callback function. [#10025]

17. Fix a race condition between DB->close and DB->{remove,rename} on filesystems that don't allow file operations on open files (such as Windows). [#10180]

18. Add a DB_DSYNC_LOG flag to the DbEnv::set_flags method, which configures O_DSYNC on POSIX systems and FILE_FLAG_WRITE_THROUGH on Win32 systems. This offers significantly better performance for some applications on certain Solaris/filesystem combinations. [#10205]

19. Fix a bug where calling the DB or DBEnv database remove or rename methods could cause a transaction checkpoint or cache flush to fail. [#10286]

20. Change file operations not to flush a file if it hasn't been written. [#10537]

21. Remove 4GB restriction on region sizes on 64 bit machines. [#10668]

22. Simplify the signature of substitute system calls for ftruncate and seek. [#10668]

23. Change Berkeley DB so that opening an environment without specifying a home directory will cause the DB_CONFIG file in the current directory to be read, if it exists. [#11424]

24. Fix a bug that caused a core dump if DB handles without associated database environments were used for database verification. [#11649]

25. Fix Windows mutexes shared between processes run as different users. [#11985]

26. Fix Windows mutexes for some SMP machines. [#12417]

Concurrent Data Store Changes:

1. Fix cursor locks for environments opened without DB_THREAD so that they use the same locker ID. This eliminates many common cases of application self-deadlock, particularly in CDS. [#9742]

General Access Method Changes:

1. Fix a bug where Berkeley DB log cursors would close and reopen the underlying log file each time the log file was read. [#8934]

2. Improve performance of DB->open() for existing subdatabases maintained within the same database file. [#9156]

3. Add a new error, DB_BUFFER_SMALL, to differentiate from ENOMEM. The new error indicates that the supplied DBT is too small. ENOMEM is now always fatal. [#9314]

4. Fix a bug when an update through a secondary index is deadlocked it is possible for the deadlock to be ignored, resulting in a partial update to the data. [#9492]

5. Fix a bug where a record could get inserted into the wrong database when a page was deallocated from one subdatabase and reallocated to another subdatabase maintained within the same database file. [#9510]

6. Enhance file allocation so that if the operating system supports decreasing the size of a file and the last page of the file is freed, it will be returned to the operating system. [#9620]

7. Fix a bug where DB_RUNRECOVERY could be returned if there was no more disk space while aborting the allocation of a new page in a database. [#9643]

8. Fix a bug where the cryptographic code could memcpy too many bytes. [#9648]

9. Fix a bug with DB->join() cursors that resulted in a memory leak and incomplete results. [#9763]

10. Disallow cursor delete, followed a cursor put of the current item across all access methods. [#9900]

11. Fix a bug where recovery of operations on unnamed databases that were never closed, could fail. [#10118]

12. Fix a bug where DB->truncate of a database with overflow records that spanned more than one page would loop. [#10151]

13. Improve performance in the database open/close path. [#10266]

14. Fix a bug that restricted the number of temporary files that could be created to 127. [#10415]

15. Fix a bug which could cause a Too many files error when trying to create temporary files. Limit the number of temporary file creation retries. [#10760] [#10773]

16. Fix a memory leak bug with Sequence Numbers. [#11589]

17. Fix a bug on Windows platforms that prevents database files from growing to over 2GB. [#11839]

18. Fix a platform independence bug with sequence numbers. Existing sequence numbers will be automatically upgraded upon next access. [#12202]

19. Fix a race between truncate and read/write operations on Windows platforms that could cause corrupt database files. [#12598]

Btree Access Method Changes:

1. Fix a bug where a record could get placed on the wrong page when two threads are simultaneously trying to split a four level (or greater) Btree. [#9542]

2. Fix a bug where calling DB->truncate() on a Btree which has duplicate keys that overflow the leaf page would not properly free the overflow pages and possibly loop. [#10666]

Hash Access Method Changes:

1. Fix a bug where a delete to a HASH database with off page duplicates could fail to have the proper lock when deleting an off page duplicate tree. [#9585]

2. Fix a bug where a dirty reader using a HASH database would leave a lock on the meta page. [#10105]

3. Fix a bug where a DB->del() on a HASH database supporting dirty reads could fail to upgrade a WWRITE lock to a WRITE lock when deleting an off page duplicate. [#10649]

Queue Access Method Changes:

1. Fix a bug where DB_CONSUME_WAIT may loop rather than wait for a new record to enter the queue, if the queue gets into a state where there are only deleted records between the head and the end of queue. [#9215]

2. Fix a bug where a Queue extent file could be closed when it was empty, even if a thread was still accessing a page from that file. [#9291]

3. Fix a bug where DBC->c_put(key, data, DB_CURRENT) where inserting a new record after the current record had been deleted was returning DB_KEYEMPTY. [#9314]

4. Fix a bug where a Queue extent file could be reported as not found if a race condition was encountered between removing the file and writing out a stale buffer. [#9462]

5. Fix a bug where the Queue access method might fail to release a record lock when running without transactions. [#9487]

6. Add DB_INORDER flag for Queue databases to guarantee FIFO (First In, First Out) ordering when using DB_CONSUME or DB_CONSUME_WAIT. [#9689]

7. Fix a bug where remove and rename calls could fail with a "Permission denied" error. [#9775]

8. Fix a bug where aborting a transaction that opened and renamed a queue database using extents could leave some of the extent files with the wrong name on Windows. [#9781]

9. Fix a bug where a db_dump of a queue database could return an error at the end of the queue if the head or tail of the queue is the first record on a page. [#10215]

10. Fix a race condition which would leave a Queue extent file open until the database handle was closed, preventing it from being removed. [#10591]

11. Fix a bug where a deadlock of a put on a database handle with dirty readers could generate a lock downgrade error. [#10678]

12. Fix a bug which caused DB_SET_RANGE and DB_GET_BOTH_RANGE to not return the next record when an exact match was not found. [#10860]

Recno Access Method Changes

1. Fix a bug where the key/data counts returned by the Db->stat method for Recno databases did not match the documentation. [#8639]

2. Fix a bug where DBC->c_put(key, data, DB_CURRENT) where inserting a new record after the current record had been deleted was returning DB_KEYEMPTY. [#9314].

C++-specific API Changes:

1. Change DbException to extend std::exception, making it possible for applications to catch all exceptions in one place. [#10022]

2. Fix a bug where errors during transaction destructors (commit, abort) could cause an invalid memory access. [#10302]

3. Fix a bug that could lead to a read through an uninitialized pointer when a DbLockNotGrantedException is thrown. [#10470]

4. Fix a bug in the C++ DbEnv::rep_elect method API where the arguments were swapped, leading to an "Invalid Argument" return when that method is called. [#11906]

Java-specific API Changes:

1. Fix a bug where the Java API did not respect non-zero return values from secondaryKeyCreate, including DB_DONOTINDEX. [#9474]

2. Fix a bug where a self-deadlock occurred with a non-transactional class catalog database used in a transactional environment. The bug only occurred when the collections API was not used for starting transactions. [#9521]

3. Improve Javadoc for the Java API. [#9614]

4. Improve memory management and performance when large byte arrays are being passed to DB methods. [#9801]

5. Improve performance of accessing statistics information from the Java API. [#9835]

6. Allow Java application to run without DB_THREAD so they can be used as RPC clients. [#10097]

7. Fix a bug where an uninitialized pointer is dereferenced for logArchive(Db.DB_ARCH_REMOVE). [#10225]

8. Fix a bug in the Collections API where a deadlock exception could leave a cursor open. [#10516]

9. Fix the replication callback in the Java API so that the parameter names match the C API. [#10550]

10. Add get methods to the Java statistics classes. [#10807]

11. Fix bugs in the Java API handling of null home directories and environments opened without a memory pool. [#11424]

12. Fix the Java API in the non-crypto package. [#11752]

13. Fix a bug that would cause corruption of error prefix strings. [#11967]

14. Fix handling of LSNs in the Java API. [#12223]

15. Dont throw a NullPointerException if the list of files returned by log_archive is empty. [#12383]

Tcl-specific API Changes:

None.

RPC-specific Client/Server Changes:

1. Add support for 64-bit and ANSI C implementations of the RPCGEN utility. [#9548]

2. Fix a small memory leak in RPC clients. [#9595]

3. Fix a bug in the RPC server to avoid self-deadlock by always setting DB_TXN_NOWAIT. [#10181]

4. Fix a bug in the RPC server so that if it times out an environment, it first closes all the Berkeley DB handles in that environment. [#10623]

Replication Changes:

1. Add an Environment ID to distinguish between clients from different replication groups. [#7786]

2. Add number of votes required and flags parameters to DB_ENV->rep_elect() method. [#7812]

3. Fix a bug where a client's env_openfiles pass could start with the wrong LSN. This could result in very long initial sync-up times for clients joining a replication group. [#8635]

4. Add election generation information to replication to support Paxos compliance. [#9068]

5. Add rep019 to test running normal recovery on clients to make sure we synch to the correct LSNs. [#9151]

6. Remove support for logs-only replication clients. Use of the DB_REP_LOGSONLY flag to the DB_ENV->rep_start() method should be replaced with the DB_REP_CLIENT flag. [#9331]

7. Fix a bug where replication clients fail to lock all the necessary pages when applying updates when there are more than one database in the transaction. [#9569]

8. Fix a bug in replication elections where when elections are called by multiple threads the wrong master could get elected. [#9770]

9. Fix a bug where the master could get a DB_REP_OUTDATED error. Instead send an OUTDATED message to the client. [#9881]

10. Add support for automatic initialization of replication clients. [#9927]

11. Modify replication timestamp so that non-replication client applications can get a DB_REP_HANDLE_DEAD. [#9986]

12. Add a new DB_REP_STARTUPDONE return value for rep_process_message() and st_startup_done to rep_stat() to indicate when a client has finished syncing up to a master and is processing live messages. [#10310]

13. Add _pp to secondary handles, add RPRINT, fix a deadlock. [#10429]

14. Fix a bug where an old client (and no master) that dropped the ALIVE message would never update to the current generation. [#10469]

15. Fix a bug where a message could get sent to a new client before NEWSITE has been returned to the application. Broadcast instead. [#10508]

16. Fix a crash when verbose replication messages are configured and a NULL DB_LSN pointer is passed to rep_process_message. [#10508]

17. Add code to respect set_rep_limit in LOG_REQ processing. [#10716]

18. Fix a synchronization problem between replication recovery and database open. [#10731]

19. Change elections to adjust timeout if egen changes while we are waiting. [#10686]

20. Client perm messages now return ISPERM/NOTPERM instead of 0. [#10855] [#10905]

21. Fix a race condition during rep_start when a role change occurs. Fix memory leaks. [#11030]

22. Fix problems with duplicate records. A failure will no longer occur if the records are old records (LOG_MORE) and archived. [#11090]

23. Fix a bug where the replication temporary database would grow during automatic client initialization. [#11090]

24. Add throttling to PAGE_REQ. [#11130]

25. Remove optimization-causing problems with racing threads in rep_verify_match. [#11208]

26. Fix memory leaks. [#11239]

27. Fix an initialization bug when High Availability configurations are combined with private database environments, which can cause intermittent failures. [#11795]

28. Fix a bug in the C++ DbEnv::rep_elect method API where the arguments were swapped, leading to an "Invalid Argument" return when that method is called. [#11906]

XA Resource Manager Changes:

None.

Locking Subsystem Changes:

1. Fix a bug where a deadlock of an upgrade from a dirty read to a write lock during an aborted transaction, may not be detected. [#7143]

2. Add support for Degree 2 isolation. [#8689]

3. Change the system to return DB_LOCK_DEADLOCK if a transaction attempts to get new locks after it has been selected as the deadlock victim. [#9111]

4. Fix a bug where when configured to support dirty reads, a writer may not downgrade a write lock as soon as possible, potentially blocking dirty readers. [#9197]

5. Change the test-and-set mutex implementation to avoid interlocked instructions when we know the instruction is unlikely to succeed. [#9204]

6. Fix a bug where a thread supporting dirty readers can get blocked while trying to get a write lock. It will allocate a new lock rather than using an existing WAS_WRITE lock when it becomes unblocked, causing the application to hang. [#10093]

7. The deadlock detector will now note that a parent transaction should be considered in abort if one of its children is. [#10394]

8. Remove a deadlock where database closes could deadlock with page acquisition. [#10726]

9. Fix a bug where a dirty reader could read an overflow page that was about to be deleted. [#10979]

10. Fix a bug that failed to downgrade existing write locks during a btree page split when supporting dirty reads. [#10983]

11. Fix a bug that would fail to upgrade a write lock when moving a cursor off a previously deleted record. [#11042]

Logging Subsystem Changes:

1. Fix a bug where recovery could leave too many files open. [#9452]

2. Fix a bug where aborting a transaction with a file open in it could result in an unrecoverable log file. [#9636]

3. Fix a bug where recovery would not return a fatal error if the transaction log was corrupted. [#9841]

4. Fix a bug in recovery so that the final checkpoint no longer tries to flush the log. This will permit recovery to complete even if there is no disk space to grow the log file. [#10204]

5. Improve performance of log flushes by pre-allocating log files and using fdatasync() in preference to fsync(). [#10228]

6. Fix a bug where recovery of a page split after a non-transactional update to the next page would fail to update the back pointer. [#10421]

7. Fix a bug in log_archive() where __env_rep_enter() was called twice. [#10577]

8. Fix a bug with in-memory logs that could cause a memory leak in the log region. [#11505]

Memory Pool Subsystem Changes:

1. Fix a bug in the MPOOLFILE file_written flag value so that checkpoint doesn't repeatedly open, flush and sync files in the cache for which there are no active application handles. [#9529]

Transaction Subsystem Changes:

1. Fix a bug where the same transaction ID could get allocated twice if you wrapped the transaction ID space twice and then had a very old transaction. [#9036]

2. Fix a bug where a transaction abort that contained a page allocation could loop if the filesystem was full. [#9461]

3. Fix implementation of DB->get_transactional() to match documentation: there is no possibility of error return, only 0 or 1. [#9526]

4. Fix a bug where re-setting any of the DB_TXN_NOSYNC, DB_TXN_NOT_DURABLE and DB_TXN_WRITE_NOSYNC flags could fail to clear previous state, potentially leading to incorrect transactional behavior in the application. [#9947]

5. Add a feature to configure the maximum number of files that a checkpoint will hold open. [#10026]

6. An aborting transaction will no longer generate an undetected deadlock. [#10394]

7. Fix a bug that prevented a child transaction from accessing a database handle that was opened by its parent transaction. [#10783]

8. Fix a bug where a checkpoint or a delayed write in another process could raise an EINVAL error if the database had been opened with the DB_TXN_NOT_DURABLE flag. [#10824]

9. Fix private transactional environments on 64-bit systems. [#11983]

Utility Changes:

1. Add debugging and performance tuning information to db_stat. Add new Berkeley DB handle methods to output debugging and performance tuning information to a C library FILE handle (C and C++ APIs only). [#9204]

2. Fix a bug where db_stat could drop core if DB->open fails and no subdatabase was specified. [#9273]

3. Add command-line arguments to the db_printlog utility to restrict the range of log file records that are displayed. [#9307]

4. Fix a bug in the locking statistics where current locks included failed lock requests. [#9314]

5. Fix a bug where db_archive would remove all log files when --enable-diagnostic and DB_NOTDURABLE were both specified. [#9459]

6. Fix a bug where db_dump with the -r flag would output extra copies of the subdatabase information. [#9808]

7. Fix a bug in db_archive that would cause log file corruption if the application had configured the environment with DB_PRIVATE. [#9841]

8. Add support in db_load for resetting database LSNs and file IDs without having to reload the database. [#9916]

9. Change the DB->stat() method to take a transaction handle as an argument, allowing DB->stat() to be called from within a transaction. [#9920]

10. Fix a bug in db_printlog where only the first filewould be displayed for in-memory logs. [#11505]

11. Fix a bug that prevented database salvage from working in Berkeley DB 4.3.21. [#11649]

12. Fix a bug in db_load which made it impossible to specify more than a single option on the command line. [#11676]

Configuration, Documentation, Portability and Build Changes:

1. Add pread and pwrite to the list of system calls applications can replace at run-time. [#8954]

2. Add support for UTF-8 encoding of filenames on Windows. [#9122]

3. Remove C++ dependency on snprintf. Compilers on HPUX 10.20 are missing header file support for snprintf(). [#9284]

4. Change Berkeley DB to not use the open system call flag O_DIRECT, unless DB configured using --enable-o_direct. [#9298]

5. Fix several problems with mutex alignment on HP/UX 10.20. [#9404]

6. Fix a memory leak when HAVE_MUTEX_SYSTEM_RESOURCES is enabled. [#9546]

7. Fix a bug in the sec002.tcl test for binary data. [#9626]

8. Fix a bug where filesystem blocks were not being zeroed out in the On-Time embedded Windows OS. [#9640]

9. Fix build problems with the Java API in Visual Studio .NET 2003. [#9701]

10. Add support for the gcc compiler on the Opteron platform. [#9725]

11. Add support for the small_footprint build option for VxWorks. [#9820]

12. Add support for linking of DLLs with MinGW. [#9957]

13. Remove the make target which builds the RPM package from the Berkeley DB distribution. [#10233]

14. Add a C++/XML example for ex_repquote. [#10380]

15. Fix a bug to link with lrt only if detected by configure (Mac OS X issue). [#10418]

16. Fix a bug and link Java and Tcl shared libraries with lpthread if required, for mutexes. [#10418]

17. Add support for building Berkeley DB on the HP NonStop OSS (Tandem) platform. [10483]

18. Change Berkeley DB to ignore EAGAIN from system calls. This fixed problems on NFS mounted databases. [#10531]

19. Remove a line with bt_compare and bt_prefix from the db_dump recovery test suite, which can cause failures on OpenBSD. [#10567]

20. Fix a conflict with the lock_init for building Berkeley DB on Cygwin. [#10582]

21. Add Unicode support for the Berkeley DB Windows API. [#10598]

22. Add support for 64-bit builds on Windows. [#10664]

23. Libtool version is now 1.5.8. [#10950]

24. Remove mt compilation flag for HP-UX 11.0. [#11427]

25. Fix a bug to link with lrt on Solaris to support fdatasync. [#11437]

Chapter 41. Upgrading Berkeley DB 4.3 applications to Berkeley DB 4.4

Release 4.4: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.3 release interfaces to the Berkeley DB 4.4 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.4: DB_AUTO_COMMIT

In previous Berkeley DB releases, the DB_AUTO_COMMIT flag was used in the C and C++ Berkeley DB APIs to wrap operations within a transaction without explicitly creating a transaction and passing the TXN handle as part of the operation method call. In the 4.4 release, the DB_AUTO_COMMIT flag no longer needs to be explicitly specified.

In the 4.4 release, specifying the DB_AUTO_COMMIT flag to the DB_ENV->set_flags() method causes all database modifications in that environment to be transactional; specifying DB_AUTO_COMMIT to the DB->open() method causes all modifications to that database to be transactional; specifying DB_AUTO_COMMIT to the DB_ENV->dbremove() methods causes those specific operations to be transactional.

No related application changes are required for this release, as the DB_AUTO_COMMIT flag is ignored where it is no longer needed. However, application writers are encouraged to remove uses of the DB_AUTO_COMMIT flag in places where it is no longer needed.

Similar changes have been made to the Berkeley DB Tcl API. These changes are not optional, and Tcl applications will need to remove the -auto_commit flag from methods where it is no longer needed.

Release 4.4: DB_DEGREE_2, DB_DIRTY_READ

The names of two isolation-level flags changed in the Berkeley DB 4.4 release. The DB_DEGREE_2 flag was renamed to DB_READ_COMMITTED, and the DB_DIRTY_READ flag was renamed to DB_READ_UNCOMMITTED, to match ANSI standard names for isolation levels. The historic flag names continue to work in this release, but may be removed from future releases.

Release 4.4: DB_JOINENV

The semantics of joining existing Berkeley DB database environments has changed in the 4.4 release. Previously:

1. Applications joining existing environments, but not configuring some of the subsystems configured in the environment when it was created, would not be configured for those subsystems.

2. Applications joining existing environments, but configuring additional subsystems in addition to the subsystems configured in the environment when it was created, would cause additional subsystems to be configured in the database environment.

In the 4.4 release, the semantics have been simplified to make it easier to write robust applications. In the 4.4 release:

1. Applications joining existing environments, but not configuring some of the subsystems configured in the environment when it was created, will now automatically be configured for all of the subsystems configured in the environment.

2. Applications joining existing environments, but configuring additional subsystems in addition to the subsystems configured in the environment when it was created, will fail, as no additional subsystems can be configured for a database environment after it is created.

In other words, the choice of subsystems initialized for a Berkeley DB database environment is specified by the thread of control initially creating the environment. Any subsequent thread of control joining the environment will automatically be configured to use the same subsystems as were created in the environment (unless the thread of control requests a subsystem not available in the environment, which will fail). Applications joining an environment, able to adapt to whatever subsystems have been configured in the environment, should open the environment without specifying any subsystem flags. Applications joining an environment, requiring specific subsystems from their environments, should open the environment specifying those specific subsystem flags.

The DB_JOINENV flag has been changed to have no effect in the Berkeley DB 4.4 release. Applications should require no changes, although uses of the DB_JOINENV flag may be removed.

Release 4.4: mutexes

The DB_ENV>set_tas_spins and DB_ENV>get_tas_spins methods have been renamed to DB_ENV->mutex_set_tas_spins() and DB_ENV->mutex_set_tas_spins() to match the new mutex support in the Berkeley DB 4.4 release. Applications calling the old methods should be updated to use the new method names.

For backward compatibility, the string "set_tas_spins" is still supported in DB_CONFIG files.

The --with-mutexalign="ALIGNMENT" compile-time configuration option has been removed from Berkeley DB configuration. Mutex alignment should now be configured at run-time, using the DB_ENV->mutex_set_align() method.

Release 4.4: DB_MPOOLFILE->set_clear_len

The meaning of a 0 "clear length" argument to the DB_MPOOLFILE->set_clear_len() method changed in the Berkeley DB 4.4 release. In previous releases, specifying a length of 0 was equivalent to the default, and the entire created page was cleared. Unfortunately, this left no way to specify that no part of the page needed to be cleared. In the 4.4 release, specifying a "clear length" argument of 0 means that no part of the page need be cleared.

Applications specifying a 0 "clear length" argument to the DB_MPOOLFILE->set_clear_len() method should simply remove the call, as the default behavior is to clear the entire created page.

Release 4.4: lock statistics

The names of two fields in the lock statistics changed in the Berkeley DB 4.4 release. The **st_nconflicts** field was renamed to be **st_lock_wait**, and the **st_nnowaits** field was renamed to be **st_lock_nowait**. The meaning of the fields is unchanged (although the documentation has been updated to make it clear what these fields really represent).

Release 4.4: Upgrade Requirements

The log file format changed in the Berkeley DB 4.4 release. No database formats changed in the Berkeley DB 4.4 release.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.4.16 Change Log

Database or Log File On-Disk Format Changes:

1. The on-disk log format has changed.

New Features:

1. Add support to compact an existing Btree database. [#6750]
2. Add support for named in-memory databases. [#9927]
3. Add support for database environment recovery serialization. This simplifies multiprocess application architectures. Add DB_REGISTER flag to DB_ENV->open(). [#11511]
4. Add utility for performing hot backups of a database environment. [#11536]
5. Add replication configuration API. [#12110]
6. Add replication support to return error instead of waiting for client sync to complete. [#12110]
7. Add replication support for delayed client synchronization. [#12110]
8. Add replication support for client-to-client synchronization. [#12110]
9. Add replication support for bulk transfer. [#12110]
10. Add new flags DB_DSYNC_DB and DB_DSYNC_LOG [12941]

11. Add DbEnv.log_printf, a new DbEnv method which logs printf style formatted strings into the Berkeley DB database environment log. [#13241]

Database Environment Changes:

1. Add a feature to support arbitrary alignment of mutexes in order to minimize cache line collisions. [#9580]

2. Change cache regions on 64-bit machines to allow regions larger than 4GB. [#10668]

3. Fix a bug where a loop could occur if the application or system failed during modification of the linked list of shared regions. [#11532]

4. Fix mutex alignment on Linux/PA-RISC, add test-and-set mutexes for MIPS and x86_64. [#11575]

5. Fix a bug where private database environments (DB_PRIVATE) on 64-bit machines would core dump because of 64-bit address truncation. [#11983]

6. Fix a bug where freed memory is accessed when DB_PRIVATE environments are closed. This can happen on systems where the operating system holds mutex resources that must be freed when the mutex is destroyed. [#12591]

7. Fix a bug where the DbEnv.stat_print method could self-deadlock and hang. The DbEnv.stat_print method no longer displays statistics for any of the database environments databases. [#12039]

8. Fix a bug where Berkeley DB could create fragmented filesystem-backed shared region files. [#12125]

9. Fix a bug where Berkeley DB stat calls could report a cache size of 0 after the statistics were cleared. [#12307]

10. Threads of control joining database environments are now configured for all of the subsystems (lock, log, cache, or transaction) for which the environment was originally configured, it is now an error to attempt configuration of additional subsystems after an environment is created. [#12422]

11. Fix a bug where negative percentages could be displayed in statistics output. [#12673]

12. Fix a bug that could cause a panic if the cache is filled with non-logging updated pages. [#12763]

13. Fix a bug that could cause an unreported deadlock if the application was using the DB_DIRTY_READ flag and the record was an off page duplicate record. [#12893]

14. Fix a bug where a handle lock could be incorrectly retained during a delete or rename operation. [#12906]

Concurrent Data Store Changes:

1. Lock upgrades and downgrades are now accounted for separately from lock requests and releases. [#11155]

2. Fix a bug where a second process joining a Concurrent Data Store environment, with the DB_CDB_ALLDB flag set, would fail. This would happen if the first thread were not entirely finished with initialization. [#12277]

General Access Method Changes:

1. Fix a bug where filesystem operations are improperly synchronized. [#10564]

2. Add support for database files larger than 2GB on Windows. [#11839]

3. Rename DB_DEGREE_2 (and all related flags) to DB_READ_COMMITTED. Rename DB_DIRTY (and all related flags) to DB_READ_UNCOMMITTED. [#11776]

4. Fix a bug where wrapping of sequences was incorrect when the cache size is smaller than the range of the maximum value minus the minimum value. [#11997]

5. Fix a bug that could result in a hot backup having a page missing from a database file if a file truncation was in progress during the backup but was then aborted. [#12017]

6. Fix a bug where a long filename could cause one too few bytes to be allocated when opening a file. [#12085]

7. Fix a bug in secondary cursor code if a write lock is not granted. [#12109]

8. Fix a bug in secondary cursors where the current record would change on error. [#12141]

9. Fix a bug in Db->truncate where the method was not checking to see if the handle was opened read-only. [#12179]

10. Fix a bug in sequences so that they are now platform independent, ta noa

Btree Access Method Changes:

1. Remove maxkey configuration. [#8904]

2. Fix a memory leak in operations on large Btrees. [#12000]

Hash Access Method Changes:

1. Fix a bug where access to HASH or encrypted database pages might be blocked during a checkpoint. [#11031]

2. Fix a bug where recovery would fail when a database has a hash page on the free list and that hash page was freed without using transactions and later allocated and aborted within a transaction. [#11214]

3. Fix a bug in hash duplicates where if the caller left garbage in the partial length field, we were using it. Fix a bug where a replacement of a hash item that should have gone on an overflow page, did not. [#11966]

4. Fix a bug where free space was miscalculated when adding the first duplicate to an existing item and the existing item plus the new item does not fit on a page. [#12270]

5. Fix a bug where allocations of hash buckets are not recovered correctly. [#12846]

Queue Access Method Changes:

1. Improve performance of deletes from a QUEUE database that does not have a secondary index. [#11538]

2. Fix a bug where updates that do not use transactions, but do enable locking, failed to release locks. [#11669]

3. Fix a bug where a transaction might not be rolled forward if the site was performing hot backups and an application aborted a prepared but not committed transaction. [#12181]

4. Fix a bug with queue extents not being reclaimed. [#12249]

5. Fix a bug where a record being inserted before the head of the queue could appear missing if DB_CONSUME is not specified. [#12919]

6. Fix a bug that might cause recovery to move the head or tail of the queue to exclude a record that was deleted but whose transaction did not commit. [#13256]

7. Fix a bug that could cause recovery to move the head or tail pointer beyond a record that was aborted but was rolled backward by recovery. [#13318]

Recno Access Method Changes

None.

C++-specific API Changes:

1. Fix a bug so that a DbMemoryException will be raised during a DB_BUFFER_SMALL error. [#13273]

Java-specific API Changes:

1. Add VersionMismatchException to map the DB_VERSION_MISMATCH error. [#11429]
2. Fix a bug in Environment.getConfiguration() method in non-crypto builds. [#11752]
3. Fix a bug that caused a NullPointerException when using the MultipleDataEntry default constructor. [#11753]
4. Fix handling of replication errors. [#11822]
5. Remove EnvironmentConfig.setReadOnly() method. [#11882]
6. Fix a bug where prefix strings in the error handler may be corrupted. [#11967]
7. Fix a bug so that nested exceptions will appear in stack traces. [#11992]
8. Fix a bug on LogSequenceNumber objects in the Java API. [#12223]
9. Fix a bug when no files are returned from a call to DB_ENV->log_archive. [#12383]
10. Fix a bug when multiple verbose flags are set. [#12383]
11. Fix a bug so that an OutOfMemoryError is thrown when allocation fails in the JNI layer. [#13434]

Java collections and bind API Changes:

1. Binding performance has been improved by using System.arraycopy in the FastOutputStream and FastInputStream utility classes. [#12002]
2. The objectToEntry method is now implemented in all TupleBinding subclasses (IntegerBinding, etc) so that tuple bindings are fully nestable. An example of this usage is a custom binding that dynamically discovers the data types of each of the properties of a Java bean class. For each property, it calls TupleBinding.getPrimitiveBinding using the property's type (class). When the custom binding's objectToEntry method is called, it in turn calls the objectToEntry method of the nested bindings for each property. [#12124]
3. The getCause method for IOExceptionWrapper and RuntimeExceptionWrapper is now defined so that nested exceptions appear in stack traces for exceptions thrown by the collections API. [#11992]
4. TupleBinding.getPrimitiveBinding can now be passed a primitive type class as well as a primitive wrapper class. The return value for Integer.TYPE and Integer.class, for example, will be the same binding. [#12035]

5. Improvements have been made to prevent the buffer used in serial and tuple bindings from growing inefficiently, and to provide more alternatives for the application to specify the desired size. For details see com.sleepycat.bind.serial.SerialBase and com.sleepycat.bind.tuple.TupleBase. [#12398]

6. Add StoredContainer.getCursorConfig, deprecate isDirtyRead. Deprecate StoredCollections.dirtyReadMap (dirtyReadSet, etc) which is replaced by configuredMap (configuredSet, etc). Deprecated StoredContainer.isDirtyReadAllowed with no replacement (please use DatabaseConfig.getDirtyRead). Also note that StoredCollections.configuredMap (configuredSet, etc) can be used to configure read committed and write lock containers, as well as read uncommitted containers, since all CursorConfig properties are supported. [#11776]

7. Add the protected method SerialBinding.getClassLoader so that subclasses may return a specific or dynamically determined class loader. Useful for applications which use multiple class loaders, including applications that serialize Groovy-defined classes. [#12764] [#12749]

Tcl-specific API Changes:

1. Fix a bug that could cause a memory leak in the replication test code. [#13436]

RPC-specific Client/Server Changes:

1. Fix double-free in RPC server when handling an out-of-memory error. [#11852]

Replication Changes:

1. Fix race condition (introduced in 4.3) in rep_start function. [#11030]

2. Changed internal initialization to no longer store records. [#11090]

3. Add support for replication bulk transfer. [#11099]

4. Berkeley DB now calls check_doreq function for MASTER_REQ messages. [#11207]

5. Fix a bug where transactions could be counted incorrectly during txn_recover. [#11257]

6. Add DB_REP_IGNORE flag so that old messages (especially PERM messages) can be ignored by applications. [#11585]

7. Fix a bug where op_timestamp was not initialized. [#11795]

8. Fix a bug in db_refresh where a client would write a log record on closing a file. [#11892]

9. Fix backward arguments in C++ rep_elect API. [#11906]

10. Fix a bug where a race condition could happen between downgrading a master and a database update operation. [#11955]

11. Fix a bug on VERIFY_REQ. We now honor wait recs/rcvd. [#12097]

12. Fix a bug in rebroadcast of verify_req by initializing lp->wait_recs when finding a new master. [#12097]

13. Fix a bug by adding lockout checking to __env_rep_enter since rename/remove now call it. [#12192]

14. Fix a bug so that we now skip __db_chk_meta if we are a rep client. [#12316]

15. Fix a replication failure on Windows. [#12331]

16. Remove master discovery phase from rep_elect as a performance improvement to speed up elections. [#12551]

17. Fix a bug to avoid multiple data streams when issuing al ALL_REQ. [#12595]

18. Fix a bug to request the remaining gap again if the gap record is dropped after we receive the singleton. [#12974]

19. Fix a bug in internal initialization when master changes in the middle of initializing. [#13074]

20. Fix a bug in replication/archiving with internal init. [#13110]

21. Fix pp handling of db_truncate. [#13115]

22. Fix a bug where rep_timestamp could be updated when it should not be updated. [#13331]

23. Fix a bug with bulk transfer when toggling during updates. [#13339]

24. Change EINVAL error return to DB_REP_JOIN_FAILURE. [#12110]

25. Add C++ exception for DB_REP_HANDLE_DEAD. [#13361]

26. Fix a bug where starting an election concurrently with processing a NEWMASTER message could cause the send function to be called with an invalid eid. [#13403]

XA Resource Manager Changes:

None.

Locking Subsystem Changes:

None.

Logging Subsystem Changes:

1. Add set_log_filemode for applications that need to set an absolute file mode on log files. [#8747]

2. Fix a bug that caused Not Found to be returned if a log file exists but is not readable. [#11185]

3. Removed checksum of records with an in-memory log buffer. [#11280]

4. Fix a bug so that the DB_LOG_INMEMORY flag can no longer be set after calling DB_ENV->open. [#11436]

5. Fix a bug introduced after release 4.0 where two simultaneous checkpoints could cause ckp_lsn values to be out of order. [#12094]

6. Fix a bug when in debug mode and using the DEBUG_ROP which will now log read operations in __dbc_logging. [#12303]

7. Fix a bug where failing to write a log record on a file close would result in a core dump later. [#12460]

8. Fix a bug where automatic log file removal, or the return of log files using an absolute path, could fail silently if the applications current working directory could not be reached using the systems getcwd library call. [#12505]

9. Avoid locking the log region if we are not going to flush the log. This can improve performance for some write-intensive application workloads. [#13090]

10. Fix a bug with a possible segment fault when memp_stat_print is called on a temporary database. [#13315]

11. Fix a bug where log_stat_print could deadlock with threads during a checkpoint. [#13315]

Memory Pool Subsystem Changes:

1. Fix a bug where modified database pages might not be flushed if recovery were run and all pages from a database were found in the system cache and up to date, followed by a system crash. [#11654]

Transaction Subsystem Changes:

1. Add new DbTxn class methods allowing applications to set/get a descriptive name associated with a transaction. The descriptive name is also displayed by the db_stat utility. [#0382]

2. Fix a bug where aborting a transaction with a large number of nested transactions could take a long time. [#10972]

3. Add support to allow the TXN_WRITE_NOSYNC flag to be specified on the transaction handle. [#11151]

4. Fix a bug that could cause a page to be on the free list twice if it was originally put on the free list by a non-transactional update and then reallocated in a transaction that aborts. [#11159]

5. Remove the requirement for the DB_AUTO_COMMIT flag to make database operations transactional. Specifying the database environment as transactional or opening the database handle transactionally is sufficient. [#11302]

6. Fix a bug so that environments created from errant programs that called dbp->close while transactions were still active can now be recovered. [#11384]

7. Fix a bug that caused free pages at the end of a file to be truncated during recovery rather than placed on the free page list. [#11643]

8. Fix a bug that caused a page to have the wrong type if the truncate of a BREE or RECNO database needed to be rolled forward. [#11670]

9. Fix a bug when manually undoing a subdb create, dont try to free a root page that has not been allocated. [#11925]

10. Add a check on database open to see if log files were incorrectly removed by system administration mistakes. [#12178]

11. Fix a bug when calling DB->pget and then specifying the DB_READ_COMMITTED (DB_DEGREE_2) on a cursor. If followed by a DBC->c_pget, the primary database would incorrectly remain locked. [#12410]

12. Fix a bug where the abort of a transaction in which a sub database was opened with the DB_TXN_NOT_DURABLE flag could fail. [#12420]

13. Fix a bug that could cause an abort transaction that allocated new pages to a file that were not flushed to disk prior to the abort transaction to report out of disk space. [#12743]

14. Fix a bug that could prevent multiple creates and destroys of the same file to be recovered correctly. [#13026]

15. Fix a bug when recovery previously handled a section of the log that did not contain any transactions. [#13139]

16. Fix a bug that could result in the loss of durability in Transactional Environments on Mac OS X. [#13149]

17. Fix a bug that could cause the improper reuse of a transaction id when recovery restores prepared transactions. [#13256]

Utility Changes:

1. Add utility for performing hot backups of a database environment. [#11536]

2. Change the Verify utility to now identify any nodes that have incorrect record counts. [#11934]

3. Fix a bug in the 1.85 compatibility code supporting per-application Btree comparison and prefix compression functions. The functions would not work on big-endian 64-bit hardware. [#13316]

Configuration, Documentation, Portability and Build Changes:

1. Change the ex_tpcb sample application to no longer displays intermediate results. It displays results at the end of the run. [#11259]

2. Change the Visual Studio projects on Windows so that each is in an intermediate directory. [#11441]

3. Fix errors in test subdb011. [#11799]

4. Fix a bug that could cause applications using gcc on Power PC platforms to hang. [#12233]

5. Fix a bug where installation will fail if a true program cannot be found. [#12278]

6. Fix a bug that prevented C++ applications from configuring XA [#12300].

7. Fix a race condition in the Windows mutex implementation found on 8-way Itanium systems. [#12417]

8. Add pthread mutex support for IBM OS/390 platform (z/OS or MVS). [#12639]

9. Fix a bug where the Tcl API did not configure on OS X 10.4. [#12699]

10. Fix portability issues with queue or recno primary databases. [#12872]

11. Fix a bug where utility attempted to send replication message. [#13446]

Berkeley DB 4.4.20 Change Log

Changes since Berkeley DB 4.4.16:

1. Add support for Visual Studio 2005. [#13521]

2. Fix a bug with in-memory transaction logs when files wrapped around the buffer. [#13589]

3. Fix a bug where we needed to close replications open files during replication initialization. [#13623]

4. Fix a bug which could leave locks in the environment if database compaction was run in a transactional environment on a non-transactional database. This might have also have triggered deadlocks if the database was opened transactionally. [#13680]

5. Fix a bug where setting the DB_REGISTER flag could result in unnecessarily running recovery, or corruption of the registry file on Windows systems. [#13789]

6. Fix a bug in Database.compact that could cause JVM crashes or NullPointerException. [#13791]

7. Fix a bug that would cause a trap if an environment was opened specifying DB_REGISTER and the environment directory could not be found. [#13793]

8. Fix a buffer overflow bug when displaying process and thread IDs in the Berkeley DB statistics output. [#13796]

9. Fix a bug where if there is insufficient memory for a database key in a DBT configured to return a key value into user-specified memory, the cursor is moved forward to the next entry in the database, which can cause applications to skip key/data pairs. [#13815]

10. Fix a bug that could cause the loss of an update to a QUEUE database in a hot backup. [#13823]

11. Fix a bug where retrieval from a secondary index could result in a core dump. [#13843]

12. Fix a bug that could cause part of the free list to become unlinked if a btree compaction was rolled back due to a transaction abort. [#13891]

13. Fix a bug with in-memory logging that could cause a race condition to corrupt the logs. [#13919]

Chapter 42. Upgrading Berkeley DB 4.4 applications to Berkeley DB 4.5

Release 4.5: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.4 release interfaces to the Berkeley DB 4.5 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.5: deprecated interfaces

Some previously deprecated interfaces were removed from the Berkeley DB 4.5 release:

- The DB_ENV->set_lk_max method was removed. This method has been deprecated and undocumented since the Berkeley DB 4.0 release.
- The DB->stat method flags DB_CACHED_COUNT and DB_RECORDCOUNT were removed. These flags have been deprecated and undocumented since the Berkeley DB 4.1 release.
- The **-w** option to the db_deadlock utility was removed. This option has been deprecated and undocumented since the Berkeley DB 4.0 release.

Release 4.5: DB->set_isalive

In previous releases, the function specified to the DB_ENV->set_isalive() method did not take a flags parameter. In the Berkeley DB 4.5 release, an additional flags argument has been added: DB_MUTEX_PROCESS_ONLY.

Applications configuring an is-alive function should add a flags argument to the function, and change the function to ignore any thread ID and return the status of just the process, when the DB_MUTEX_PROCESS_ONLY flag is specified.

Release 4.5: DB_ENV->rep_elect

Two of the historic arguments for the DB_ENV->rep_elect() method have been moved from the interface to separate methods in order to make them available within the new replication manager framework.

The **priority** parameter should now be explicitly set using the DB_ENV->rep_set_priority() method. To upgrade existing replication applications to the Berkeley DB 4.5 DB_ENV->rep_elect() interface, it may be simplest to insert a call to DB_ENV->rep_set_priority() immediately before the existing call to DB_ENV->rep_elect(). Alternatively, it may make more sense to add a single call to DB_ENV->rep_set_priority() during database environment configuration.

The **timeout** parameter should now be explicitly set using the DB_ENV->rep_set_timeout() method. To upgrade existing replication applications to the Berkeley DB 4.5 DB_ENV->rep_elect() interface, it may be simplest to insert a call to DB_ENV->rep_set_timeout() immediately before

the existing call to DB_ENV->rep_elect(). Alternatively, it may make more sense to add a single call to DB_ENV->rep_set_timeout() during database environment configuration.

Release 4.5: Replication method naming

The method names DB_ENV->set_rep_limit, DB_ENV->get_rep_limit and DB_ENV->set_rep_transport have been changed to DB_ENV->rep_set_limit(), DB_ENV->rep_get_limit() and DB_ENV->rep_set_transport() in order to be consistent with the other replication method names. That is, the characters "set_rep" and "get_rep" have been changed to "rep_set" and "rep_get".

Applications should modify the method names, no other change is required.

Release 4.5: Replication events

One of the informational returns from the DB_ENV->rep_process_message() method found in previous releases of Berkeley DB has been changed to an event. The DB_REP_STARTUPDONE return from DB_ENV->rep_process_message() is now the DB_EVENT_REP_STARTUPDONE value to the DB_ENV->set_event_notify() callback.

Applications should update their handling of this event as necessary.

Release 4.5: Memory Pool API

As part of implementing support for multi-version concurrency control, the DB_MPOOL_DIRTY flag is now specified to the DB_MPOOLFILE->get() instead of DB_MPOOLFILE->put(), and the DB_MPOOLFILE->set method has been removed. In addition, a new transaction handle parameter has been added to the DB_MPOOLFILE->get() method.

The DB_MPOOL_CLEAN flag is no longer supported.

Applications which use the memory pool API directly should update to the new API in order to use 4.5.

Release 4.5: DB_ENV->set_paniccall

In previous Berkeley DB releases, the DB_ENV->set_paniccall and DB->set_paniccall methods were used to register a callback function, called if the database environment failed. In the 4.5 release, this functionality has been replaced by a general-purpose event notification callback function, set with the DB_ENV->set_event_notify() method. Applications should be updated to replace DB_ENV->set_paniccall and DB->set_paniccall calls with a call to DB_ENV->set_event_notify(). This also requires the callback function itself change, as the callback signatures are different.

The DB_ENV->set_paniccall and DB->set_paniccall calls are expected to be removed in a future release of Berkeley DB.

Release 4.5: DB->set_pagesize

In previous releases, when creating a new database in a physical file which already contained databases, it was an error to specify a page size different from the existing databases in the file. In the Berkeley DB 4.5 release, any page size specified is ignored if the file in which the database is being created already exists.

Release 4.5: Collections API

The changes to the Collections API are compatible with prior releases, with one exception: the Iterator object returned by the StoredCollection.iterator() method can no longer be explicitly cast to StoredIterator because a different implementation class is now used for iterators. If you depend on the StoredIterator class, you must now call StoredCollection.storedIterator() instead. Note the StoredIterator.close(Iterator) static method is compatible with the new iterator implementation, so no changes are necessary if you are using that method to close iterators.

Release 4.5: --enable-pthread_self

In previous releases, the --enable-pthread_self configuration option was used to force Berkeley DB to use the POSIX pthread pthread_self function to identify threads of control (even when Berkeley DB was configured for test-and-set mutexes). In the 4.5 release, the --enable-pthread_self option has been replaced with the --enable-pthread_api option. This option has the same effect as the previous option, but configures the Berkeley DB build for a POSIX pthread application in other ways (for example, configuring Berkeley DB to use the pthread_self function).

Release 4.5: Recno backing text source files

In previous releases of Berkeley DB, Recno access method backing source text files were opened using the ANSI C fopen function with the "r" and "w" modes. This caused Windows systems to translate carriage-return and linefeed characters on input and output and could lead to database corruption.

In the current release, Berkeley DB opens the backing source text files using the "rb" and "wb" modes, consequently carriage-return and linefeed characters will not be translated on Windows systems.

Applications using the backing source text file feature on systems where the "r/w" and "rb/wb" modes differ should evaluate their application as part of upgrading to the 4.5 release. There is the possibility that characters have been translated or stripped and the backing source file has been corrupted. (Applications on other systems, for example, POSIX-like systems, should not require any changes related to this issue.)

Release 4.5: Application-specific logging

In previous releases of Berkeley DB, "BEGIN" lines in the XXX.src files used to build application-specific logging support only required a log record number. In the 4.5 release, those lines require a Berkeley DB library version as well. For example, the entry:

```
BEGIN mkdir 10000
```

must now be:

```
BEGIN mkdir 44 10000
```

that is, the version of the Berkeley DB release where the log record was introduced must be included. The version is the major and minor numbers for the Berkeley DB library, with all punctuation removed. For example, Berkeley DB version 4.2 should be 42, version 4.5 should be 45.

Release 4.5: Upgrade Requirements

The log file format changed in the Berkeley DB 4.5 release. No database formats changed in the Berkeley DB 4.5 release.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.5.20 Change Log

Database or Log File On-Disk Format Changes:

1. The on-disk log format has changed.

New Features:

1. Multi-Version Concurrency Control for the Btree/Recno access methods.
2. A new replication framework with a default TCP/IP setup.
3. Online replication upgrades for high availability replicated 24/7 systems.
4. A new event-style notification.
5. Several enhancements to the Java Collections API including the implementation of the size() method.

Database Environment Changes:

1. Update the DB_ENV->failchk method to garbage collect per-process mutexes stranded after unexpected process failure. [#13964]
2. Fix a bug that could cause memory used to track threads for DB_ENV->failchk to not be reused when a thread no longer exists. [#14425]

3. Add set_event_notify behavior as part of new event notification in Berkeley DB. [#14534]

4. Fix a bug so that we no longer panic on DB_ENV->close() if a previous environment close failed to log. This condition will now return an error. [#14693]

5. Created os_getenv, removed clib/getenv, implemented Windows specific behavior. [#14942]

6. Fix a bug where it was possible to corrupt the DB_REGISTER information file, making it impossible for Berkeley DB applications to join database environments. [#14998]

Concurrent Data Store Changes:

1. Fix a bug where renaming a subdatabase in a Concurrent Data Store environment could fail. [#14185]

General Access Method Changes:

1. Fix a bug that could leave extra unallocated pages at the end of a database file. [#14031]

2. Optimize secondary updates when overwriting primary records. [#14075]

3. Fix a bug to prevent a trap when creating a named in-memory database and there are already temporary files open. [#14133]

4. Fix a bug which caused a trap if the key parameter to DBC->c_get was omitted with DB_CURRENT. [#14143]

5. Fix a bug with secondary cursors when the secondary has off-page duplicates. This bug resulted in incorrect primary data being returned. [#14240]

6. Improve performance when removing a subdatabase by not locking every page. [#14366]

7. Fix a bug that would not properly upgrade database files from releases 3.2.9 (and earlier) to releases 4.0 (and greater). [#14461]

8. Fix a bug that could cause a DB_READ_UNCOMMITTED get through a secondary index to return DB_SECONDARY_CORRUPT. [#14487]

9. Fix a bug so that non-transactional cursor updates of a transactional database will generate an error. [#14519]

10. Add a message when the system panics due to a page in the wrong state at its time of allocation. [#14527]

11. Fix a remove failure when attempting to remove a file that is open in another thread of control. [#14780]

12. Fix a bug where the key was not ignored when doing a cursor put with the DB_CURRENT flag. [#14988]

Btree Access Method Changes:

a. When deleting a page don't check the next key in the parent if we are going to delete the parent too.

b. Need to check that the tree has not collapsed between dropping a read lock and getting the write lock. If it has collapsed we will fetch the root of the tree.

c. Fix a case where we fail to lock the next page before reading it.

1. Changed the implementation of internal nodes in btrees so that they no longer share references to overflow pages with leaf nodes. [#10717]

2. Fix a bug that could cause a diagnostic assertion by setting the deleted bit on a record in an internal node. [#13944]

3. Fix three problems in BTREE compaction: [#14238]

4. Fix a bug that could cause the compaction of a Btree with sorted duplicates to fail when attempting to compact an off page duplicate tree if a key could not fit in an internal node. [#14771]

5. Fix a bug that causes a loop if an empty Btree was compacted. [#14493]

Hash Access Method Changes:

1. Fix a bug to allow creation of hash pages during truncate recovery. [#14247]

Queue Access Method Changes:

1. Fix a bug where reads of data items outside the range of the queue were not kept locked to the end of the transaction, breaking serializability. [#13719]

2. Fix a bug that could cause corruption in queue extent files if multiple processes tried to open the same extent at the same time. [#14438]

3. Improve concurrency for in-place updates in queue databases. [#14918]

Recno Access Method Changes:

None.

C++-specific API Changes:

1. C++ applications that check the error code in exceptions should note that DbMemoryException has been changed to have the error code DB_BUFFER_SMALL rather than ENOMEM, to match the error returned by the C API. DbMemoryException will be thrown when a Dbt is too small to contain data returned by Berkeley DB. When a call to malloc fails, or some other resource is exhausted, a plain DbException will be thrown with error code set to ENOMEM. [#13939]

Java-specific API Changes:

1. Database.verify may now be called. This method is now static and takes a DatabaseConfig parameter. [#13971]

2. Add DB_ENV->{fileid_reset, lsn_reset} to the public API. [#14076]

Java collections and bind API Changes:

1. The com.sleepycat.collections package is now fully compatible with the Java Collections framework. [#14732]

Tcl-specific API Changes:

1. Fix a conflicting variable, sysscript.tcl. [#15051]

RPC-specific Client/Server Changes:

None.

Replication Changes:

1. Fix a bug when running with DEBUG_ROP or DEBUG_WOP. [#13394]

2. Add live replication upgrade support [#13670]

3. Fix a bug so that client databases are removed at the start of internal initialization. [#14147]

4. Fix a bug in replication internal initialization so that data_dir will be handled correctly. Make internal initialization resilient to multiple data_dir calls with the same directory. [#14489]

5. Fix a bug in the 4.2 sync-up algorithm that could result in no open files. [#14552]

6. Fix a bug when clients decide to re-request. [#14642]

7. Fix a bug where a PERM bulk buffer could have a zero LSN passed to the application callback. [#14675]

8. Change names of some existing replication API methods as described in "Replication Method Naming" page of the "Upgrading Berkeley DB Applications to Release 4.5" section of Berkeley DB Reference Guide. [#14723]

9. Fix a bug which could cause an election to succeed only after waiting for the timeout to expire, even when all sites responded in a timely manner. The bug was most easily visible in an election between 2 sites. [#14752]

10. Fix a bug where a process could have an old file handle to a log file. [#14797]

11. Fix a bug where a "log_more" message could be on a log file boundary. [#15034]

12. Fix a bug that could cause log corruption if a database open operation were attempted during a call to rep_start in another thread. [#15035]

13. Fix a bug during elections where a vote2 arrives before its vote1. [#15055]

14. Fix a bug to make sure we are a client if sending a REP_REREQUEST. [#15066]

XA Resource Manager Changes:

None.

Locking Subsystem Changes:

1. Fix a bug that could cause a write to hang if DB_READ_UNCOMMITTED is enabled and it tries to reacquire a write lock. [#14919]

Logging Subsystem Changes:

1. Fix a bug so that log headers are now included in the checksum. This avoids a possible race in doing hot backups. [#11636].

2. Add a check so that some log sequence errors are diagnosed at run time rather than during recovery. [#13231]

3. Fix a bug where recovery fails if there is no disk space for the forced checkpoint that occurs at the end of processing the log. [#13986]

4. Fix a bug which could cause a page to be missing from the end of a database file if the page at the end of the file was freed while it contained data and the system was restarted before the log record for that free was flushed to disk. [#14090]

5. Fix a bug that could cause log files to be incorrectly removed by log_archive if it was run immediately after recovery. [#14874]

Memory Pool Subsystem Changes:

1. Fix a bug that could cause corruption to the buffer pool cache if a race condition was hit while using DB->compact. [#14360]

2. Fix a bug where cache pages could be leaked in applications creating temporary files for which the DB_MPOOL_NOFILE flag was set. [#14544]

Transaction Subsystem Changes:

1. Fix a bug that could cause extra empty pages to appear in a database file after recovery. [#11118]

2. Fix a bug triggered when running recovery with a feedback function that could cause a NULL pointer dereference. [#13834]

3. Fix a bug where running recovery could create duplicate entries in the data directory list. [#13884]

4. Fix a bug to not trade locks if a write lock is already owned. [#13917]

5. Fix a bug that could cause traps or hangs if the DB_TXN->set_name function is used in a multithreaded application. [#14033]

6. Fix a bug so that a transaction can no longer be committed after it had deadlocked. [#14037]

7. Fix a bug that could cause a trap during recovery if multiple operations that could remove the same extent are recovered. [#14061]

8. Fix a bug that could cause an extent file to be deleted after the last record in the extent was consumed but the consuming transaction was aborted. [#14179]

9. Fix a bug where the parent database would not use DB_READ_UNCOMMITTED in certain cases when calling DBC->c_pget. [#14361]

10. Fix a bug so that it is no longer possible to do a non-transactional cursor update on a database that is opened transactionally. [#14519]

11. Fix a bug that causes a sequence to ignore the DB_AUTO_COMMIT settings. [#14582]

12. Fix a bug, change txn_recover so that multiple processes will recover prepared transactions without requiring that the first process stay active. [#14707]

13. Fix a bug that could cause the wrong record to be deleted if a transaction had a cursor on a record with a pending delete and then replaced a record that contained overflow data or replaced a record with overflow data and that replace failed. [#14834]

Utility Changes:

1. Fix a bug that caused db_verify to not check the order on leaf pages which were the leftmost children of an internal node. [#13004]

2. Fix a bug that caused db_hotbackup to not backup queue extent files. [#13848]

3. Fix a bug so that db_verify no longer reports that an unused hash page is not fully zeroed. [#14030]

4. Fix a bug where db_stat ignored the -f option to return "fast statistics". [#14283]

5. Fix a bug that prevented the db_stat utility from opening database files with write permission so that meta data statistics would be updated. [#14755]

6. Fix a bug in db_hotbackup related to windows. Sub-directories are now ignored. [#14757]

Configuration, Documentation, Portability and Build Changes:

1. The Berkeley DB 4.3 and 4.4 releases disallowed using the --with-uniquename configuration option with the C++, Java, or RPC --enable-XXX options. The 4.5 release returns to the 4.2 release behavior, allowing those combinations of configuration options. [#14067]

2. Fix build issues when CONFIG_TEST is not enabled for Tcl. [#14507]

3. There are updated build instructions for Berkeley DB PHP module on Linux. [#14249]

4. Use libtool's "standard" environment variable names so that you can set "AR" to "ar -X64" for example, and modify both libtool and the Makefile commands. Remove the install-strip target from the Makefile, it is no longer used. [#14726]

5. Fix a bug where, when a database is opened with the DB_THREAD flag (the default in Java), and an operation in one thread causes the database to be truncated (typically when the last page in the database is freed) concurrently with a read or write in another thread, there can be arbitrary data loss, as Windows zeros out pages from the read/write location to the end of the file. [#15063]

Chapter 43. Upgrading Berkeley DB 4.5 applications to Berkeley DB 4.6

Release 4.6: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.5 release interfaces to the Berkeley DB 4.6 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.6: C API cursor handle method names

In the Berkeley DB 4.6 release, the C API DBC handle methods have been renamed for consistency with the C++ and Java APIs. The change is the removal of the leading "c_" from the names, as follows:

DBC->c_close
: Renamed DBC->close

DBC->c_count
: Renamed DBC->count

DBC->c_del
: Renamed DBC->del

DBC->c_dup
: Renamed DBC->dup

DBC->c_get
: Renamed DBC->get

DBC->c_pget
: Renamed DBC->pget

DBC->c_put
: Renamed DBC->put

The old DBC method names are deprecated but will continue for work for some number of future releases.

Release 4.6: DB_MPOOLFILE->put

The DB_MPOOLFILE->put() method takes a new parameter in the Berkeley DB 4.6 release, a page priority. This parameter allows applications to specify the page's priority when returning the page to the cache.

Applications calling the DB_MPOOLFILE->put() method can upgrade by adding a DB_PRIORITY_UNCHANGED parameter to their calls to the DB_MPOOLFILE->put() method. This will result in no change in the application's behavior.

Release 4.6: DB_MPOOLFILE->set

The DB_MPOOLFILE->set method has been removed from the Berkeley DB 4.6 release. Applications calling this method can upgrade by removing all calls to the method. This will result in no change in the application's behavior.

Release 4.6: Replication Events

It is now guaranteed the DB_EVENT_REP_STARTUPDONE event will be presented to the application after the corresponding DB_EVENT_REP_NEWMASTER event, even in the face of extreme thread-scheduling anomalies. (In previous releases, if the thread processing the NEWMASTER message was starved, and STARTUPDONE occurred soon after, the order might have been reversed.)

In addition, the DB_EVENT_REP_NEWMASTER event is now presented to all types of replication applications: users of either the Replication Framework or the Base Replication API. In both cases, the DB_EVENT_REP_NEWMASTER event always means that a site other than the local environment has become master.

The **envid** parameter to DB_ENV->rep_process_message() has been changed to be of type "int" rather than "int *", and the environment ID of a new master is presented to the application along with the DB_EVENT_REP_NEWMASTER event. Replication applications should be modified to use the DB_EVENT_REP_NEWMASTER event to determine the ID of the new master.

The **envid** parameter has been removed from the DB_ENV->rep_elect() method and a new event type has been added. The DB_EVENT_REP_ELECTED event is presented to the application at the site which wins an election. In the Berkeley DB 4.6 release, the normal result of a successful election is either the DB_EVENT_REP_NEWMASTER event (with the winner's environment ID), or the DB_EVENT_REP_ELECTED event. Only one of the two events will ever be delivered.

The DB_REP_NEWMASTER return code has been removed from the DB_ENV->rep_process_message() method. Replication applications should be modified to use the DB_EVENT_REP_NEWMASTER and DB_EVENT_REP_ELECTED events to determine the existence of a new master.

Release 4.6: DB_REP_FULL_ELECTION

The DB_REP_FULL_ELECTION flag historically specified to the DB_ENV->repmgr_start() method has been removed from the 4.6 release.

In the Berkeley DB 4.6 release, a simpler and more flexible implementation of this functionality is available. Applications needing to configure the first election of a replication group differently from subsequent elections should use the DB_REP_FULL_ELECTION_TIMEOUT flag to the DB_ENV->rep_set_timeout() method to specify a different timeout for the first election.

Release 4.6: Verbose Output

When an error occurs in the Berkeley DB library, an exception is thrown or an error return value is returned by the interface. In some cases, however, the exception or returned value may be insufficient to completely describe the cause of the error, especially during initial application debugging. Applications can configure Berkeley DB for verbose messages to be output when an error occurs, but it's a common cause of confusion for new users that no verbose messages are available by default.

In the Berkeley DB 4.6 release, verbose messages are configured by default. For the C and C++ APIs, this means the default configuration when applications first create DB or DB_ENV handles is as if the DB_ENV->set_errfile() or DB->set_errfile() methods were called with the standard error output (stderr) specified as the FILE * argument. Applications wanting no output at all can turn off this default configuration by calling the DB_ENV->set_errfile() or DB->set_errfile() methods with NULL as the FILE * argument. Additionally, explicitly configuring the error output channel using any of the DB_ENV->set_errfile(), DB_ENV->set_errcall(), DbEnv::set_error_stream() or Db::set_error_stream() methods will also turn off this default output for the application.

Applications which configure Berkeley DB with any error output channel should not require any changes.

Applications which depend on having no output from the Berkeley DB library by default, should be changed to call the DB_ENV->set_errfile() or DB->set_errfile() methods with NULL as the FILE * argument.

Release 4.6: DB_VERB_REPLICATION

The DB_VERB_REPLICATION flag no longer requires the Berkeley DB library be built with the --enable-diagnostic configuration option to output additional replication logging information.

Release 4.6: Windows 9X

Berkeley DB no longer supports process-shared database environments on Windows 9X platforms; the DB_PRIVATE flag must always be specified to the DB_ENV->open() method.

Release 4.6: Upgrade Requirements

The log file format changed in the Berkeley DB 4.6 release.

The format of Hash database pages was changed in the Berkeley DB 4.6 release, and items are now stored in sorted order. **The format changes are entirely backward-compatible, and no database upgrades are needed.** However, upgrading existing databases can offer significant performance improvements. Note that databases created using the 4.6 release may not be usable with earlier Berkeley DB releases.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.6.21 Change Log

4.6.21 Patches:

1. Fix a bug where mutex contention in database environments configured for hybrid mutex support could result in performance degradation. [#15646]

2. Fix a bug where closing a database handle after aborting a transaction which included a failed open of that database handle could result in application failure. [#15650]

3. Fix multiple MVCC bugs including a race which *could result in incorrect data being returned* to the application. [#15653]

4. Fix a bug where a database store into a Hash database could self-deadlock in a database environment configured for the Berkeley DB Concurrent Data Store product and with a free-threaded DB_ENV or DB handle. [#15718]

5. Fix an installation bug where Berkeley DB's PHP header file was not installed in the correct place.

4.6.19 Patches

1. Fix a bug where a client in a two-site replication group could become master, after failure of the existing master, even if the client had priority 0. [#15388]

2. Fix a bug where 32-bit builds on 64-bit machines could immediately core dump because of a misaligned access. [#15643]

3. Fix a bug where attempts to configure a database for MVCC in the Java API were silently ignored. [#15644]

4. Fix a bug where database environments configured for replication and verbose output could drop core. [#15651]

Database or Log File On-Disk Format Changes:

1. The on-disk log format has changed.

2. The format of Hash database pages was changed in the Berkeley DB 4.6 release, and items are now stored in sorted order. *The format changes are entirely backward-compatible, and no database upgrades are needed.* However, upgrading existing databases can offer significant performance improvements. Note that databases created using the 4.6 release may not be usable with earlier Berkeley DB releases.

New Features:

1. Add support for a cursor DB_PREV_DUP flag, which moves the cursor to the previous key/data pair if it's a duplicate of the current key/data pair. [#4801]

2. Add the ability to set cache page priority on a database or cursor handle. [#11886]

3. Add verbose output tracing for filesystem operations. [#13760]

4. Port Berkeley DB to Qualcomm's Binary Runtime Environment for Wireless (BREW). [#14562]

5. Port Berkeley DB to WinCE. [#15312]

6. Port Berkeley DB to S60. [#15371]

7. Add a key_exists method to the DB handle. [#15374]

8. Applications may now begin processing new transactions while previously prepared, but unresolved, transactions are still pending. [#14754]

9. Significant performance improvements in the Hash access method. [#15017]

Database Environment Changes:

1. Add support to close open file handles in the case of catastrophic database environment failure so applications that do not exit and restart on failure won't leak file handles. [#6538]

2. Replace the Berkeley DB shared memory allocator with a new implementation, intended to decrease the performance drop-off seen in database environments having working sets that are larger than the cache, especially database environments with multiple cache page sizes. [#13122]

3. Fix a bug that would incorrectly cause a thread to appear to be in the Berkeley DB API after a call to db_create. [#14562]

4. Allow database close prior to resolving all transactions updating the database. [#14785]

5. Fix a bug where the db_stat utility -Z flag and the statistics method's DB_STAT_CLEAR flag could clear mutex statistics too quickly, leading to incorrect values being displayed. [#15032]

6. Fix a bug where removal of a file after and open/close pair spanning the most recent checkpoint log-sequence-numbers made recovery fail. [#15092]

7. Fix a bug that could leave an environment unrecoverable if FTRUNCATE was not set and a roll-forward to a timestamp was interrupted between the truncation of the log and the recording of aborted allocations. [#15108]

8. Fix a bug where recovery of a rename operation could fail if the rename occurred in a directory that no longer existed. [#15119]

9. Fix a bug that could cause recovery to report a "File exists" error if a committed create was partially recovered by a previously failed recovery operation. [#15151]

10. Fix a bug where the DbEnv.get_thread_count method implementation was missing from the Berkeley DB 4.5 release. [#15201]

11. Fix a bug where replication operations were not reported properly when the DbEnv.failchk method was called. [#15094]

12. Fixed a bug that caused SEQ->remove not to use a transaction if the sequence was opened on a transactional database handle but no transaction was specified on the call. [#15235]

13. Fix a bug where accesses to the database environment reference count could race, causing the DB_ENV->remove method to incorrectly remove or not remove a database environment. [#15240]

14. Fix a bug that could cause a recovery failure if a partial record was written near the end of a log file before a crash and then never overwritten after recovery runs and before a log file switch occurs. [#15302]

15. Fix a bug that could fire a diagnostic assertion if an error occurred during a database environment open. [#15309]

16. Fix a bug where memp_trickle attempts to flush an infinite number of buffers. [#15342]

17. Cause application updates of the DB_ENV->set_mp_max_write values to affect already running cache flush operations. [#15342]

18. Fix a bug which could cause system hang if a checkpoint happened at the same time as a database file create or rename. [#15346]

19. Fix a bug which could cause application failure if the open of a subdatabase failed while other database opens were happening. [#15346]

20. Fix a bug that could cause recovery to not process a transaction properly if the transaction was started before the transaction IDs were reset but did not put its first record into the log until after the txn_recycle record. [#15400]

21. Fix a bug that could cause a thread in cache allocation to loop infinitely. [#15406]

22. Fix a bug that could cause recovery to report a Log Sequence Error on systems without the ftruncate system call where a page allocation occurred and the database metadata page was forced out of cache without being marked dirty and then had to be recovered. [#15441]

23. Fix a bug on systems lacking the ftruncate system call, where a page may be improperly linked into the free list if archive recovery was done in multiple steps, that is, applying additional logs to the same databases. [#15557]

Concurrent Data Store Changes:

None.

General Access Method Changes:

1. Add a feature where applications can specify a custom comparison function for the Hash access method [#4109]

2. Open, create, close and removal of non-transactional databases is are longer logged in transactional database environments unless debug logging is enabled. [#8037]

3. Add the ability to set cache page priority on a database or cursor handle. [#11886]

4. fix a bug where the DB_ENV->fileid_reset method failed when called on on encrypted or check-summed databases. [#13990]

5. Fix a bug where the DB->fd method failed when called on in-memory databases. [#14157]

6. Fix a bug where an attempt to open a Recno database with a backing file that does not exist could report an error because it couldn't remove a temporary file. [#14160]

7. Reverse a change found in previous releases which disallowed setting "partial" flags on key DBTs for DB and DbCursor put method calls. [#14520]

8. Fix a bug where transactional file operations, such as remove or rename, could leak file handles. [#15222]

9. Fix a bug that could cause the in-memory sorted freelist used by the DB->compact method not to be freed if transaction or lock timeouts were set in the environment. [#15292]

10. Add the DB->get_multiple method, which returns if the DB handle references a "master" database in the physical file. [#15352]

11. Fix a bug that could cause an DB_INORDER, DB->get method DB_CONSUME operation to loop if the Queue database was missing a record due to a rollback by a writer or a non-queue insert in the queue. [#15452]

12. Fix a bug preventing database removal after application or system failure in a database environment configured for in-memory logging. [#15459]

Btree Access Method Changes:

None.

Hash Access Method Changes:

1. Change the internal format of Hash database pages, storing items in sorted order. There are no externally visible changes, and hash databases using historic on-page formats do not require an explicit upgrade. (However, upgrading existing databases can offer significant performance improvements.) [#15017]

2. Fix a bug preventing LSNs from being reset on hash databases when the databases were configured with a non-standard hash function. [#15567]

Queue Access Method Changes:

1. Fix a bug which could cause a Queue extent file to be incorrectly removed if an empty extent file was being closed by one thread and being updated by another thread (which was using random access operations). [#9101]

Recno Access Method Changes:

None.

C++-specific API Changes:

None.

Java-specific API Changes:

1. Add a feature where an exception is thrown by the Java API, the Berkeley DB error message is now included in the exception object. [#11870]

2. Fix a bug which can cause a JVM crash when doing a partial get operation. [#15143]

3. Fix a bug which prevented the use of Berkeley DB sequences from Java. [#15220]

4. Fix multiple bugs where DBTs were not being copied correctly in the Java replication APIs. [#15223]

5. Add transaction.commitWriteNoSync to the Java API. [#15376]

Java collections and bind API Changes:

1. Change SerialBinding to use the current thread's context class loader when loading application classes. This allows the JE jar file to be deployed in application servers and other containers as a shared library rather than as an application jar. [#15447]

2. Tuple bindings now support the java.math.BigInteger type. Like other tuple binding values, BigInteger values are sorted in natural integer order by default, without using a custom comparator. For details please see the Javadoc for:
com.sleepycat.bind.tuple.TupleInput.readBigInteger
com.sleepycat.bind.tuple.TupleOutput.writeBigInteger
com.sleepycat.bind.tuple.BigIntegerBinding [#15244]

3. Add tuple binding methods for reading and writing packed int and long values. Packed integer values take less space, but take slightly more processing time to read and write. See: TupleInput.readPackedInt TupleInput.getPackedIntByteLength TupleInput.readPackedLong TupleInput.getPackedLongByteLength TupleOutput.writePackedInt TupleOutput.writePackedLong PackedInteger [#15422]

4. The Collections API has been enhanced so that auto-commit works for the standard Java Iterator.remove(), set() and add() methods. Previously it was necessary to explicitly begin and commit a transaction in order to call these methods, when the underlying Database was transactional. Note that starting a transaction is still necessary when calling these methods if the StoredCollection.storedIterator method is used. [#15401]

5. Fix a bug that causes a memory leak for applications where both of the following are true: many Environment objects are opened and closed, and the CurrentTransaction or TransactionRunner class is used. [#15444]

Tcl-specific API Changes:

None.

RPC-specific Client/Server Changes:

None.

Replication Changes:

1. Fix a bug where transactions could be rolled-back if an existing replication group master was partitioned and unable to participate in an election. [#14752]

2. Add a new event when a replication manager framework master fails to send and confirm receipt by clients of a "permanent" message. [#14775]

3. Fix a race where multiple threads might attempt to process a LOGREADY condition. [#14902]

4. Change the DB_VERB_REPLICATION flag to no longer require the Berkeley DB library be built with the --enable-diagnostic configuration option to output additional replication logging information. [#14991]

5. Fix a bug with elections occurring during internal init of a replication client site. [#15057]

6. Fix lockout code to lockout message threads and API separately. Send indication that log requests is for internal init. [#15067]

7. Replication manager changed to retry host-name look-up failures, since they could be caused by transient name server outage. [#15081]

8. Fix a bug which led to memory corruption when the sending of a bulk buffer resulted in an error. [#15100]

9. A throttling limit of 10 megabytes is now set by default in a newly created database environment (see the DbEnv.rep_set_limit method). [#15115]

10. Fix a bug in ALL_REQ handling where master could get a DB_NOTFOUND. [#15116]

11. Fix a bug which could lead to client sites repeatedly but unproductively calling for an election, when a master site already exists. [#15128]

12. Modify gap processing algorithms so XXX_MORE messages ask for data beyond what it just processed, not an earlier gap that might exist. [#15136]

13. Fixed a bug in the ex_rep example application which could cause the last few transactions to disappear when shutting down the sites of the replication group gracefully. [#15162]

14. Fix a bug where if a client crashed during internal init, its database environment would be left in a confused state, making it impossible to synchronize again with the master. [#15177]

15. Fix a bug where election flags are not cleared atomically with the setting of the new master ID. [#15186]

16. Fix a bug which would cause Berkeley DB to crash if an internal init happened when there were no database files at the master. [#15227]

17. It is now guaranteed that the DB_EVENT_REP_STARTUPDONE event will be presented to the application after the corresponding DB_EVENT_REP_NEWMASTER event, even in the face of extreme scheduling anomalies. [#15265]

18. Fix minor memory leaks in the replication manager. [#15239] [#15256]

19. Fix a bug which caused the replication manager to lose track of a failed connection, resulting in the inability to accept a replacement connection. [#15311]

20. Fix a bug where a client starting an election when the rest of the replication group already had an established master could confuse replication management at the other client sites, leading to failure to properly acknowledge PERM transactions from the master. [#15428]

21. Add support for reporting Replication Manager statistics. [#15430]

22. Fix a bug where a send failure during processing of a request message from a client could erroneously appear to the application as an EPERM system error. [#15436]

23. Client now sets STARTUPDONE at the end of the synchronization phase when it has caught up to the end of the master's transaction log, without requiring ongoing transactions at the master. [#15542]

24. Fix a bug in sleep-time calculation which could cause a Replication Manager failure. [#15552]

XA Resource Manager Changes:

None.

Locking Subsystem Changes:

1. Change the DB_ENV->lock_detect method to return the number of transactions timed out in addition to those were rejected due to deadlock. [#15281]

Logging Subsystem Changes:

None.

Memory Pool Subsystem Changes:

1. Fix a bug that could cause a checkpoint to hang if a database was closed while the checkpoint was forcing that file to disk and all the pages for that database were replaced in the cache. [#15135]

2. Fix a bug where a system error in closing a file could result in a core dump. [#15137]

3. Fix MVCC statistics counts for private database environments. [#15218]

Transaction Subsystem Changes:

1. Fix a bug where creating a database with the DB_TXN_NOTDURABLE flag set would still write a log record. [#15386]

2. Change transaction checkpoint to wait only for pages being updated during the checkpoint. [#14710]

Utility Changes:

1. Fix a bug that prevented db_load from handling subdatabase names that were of zero length. [#8204]

2. Fix a bug where the db_hotbackup utility did not clean out and record the log file numbers in the backup directory when both the -u and -D flags were specified. [#15395]

Configuration, Documentation, Portability and Build Changes:

1. Berkeley DB no longer supports process-shared database environments on Windows 9X platforms; the DB_PRIVATE flag must always be specified to the DB_ENV->open method. [#13766]

2. Port Berkeley DB to Qualcomm's Binary Runtime Environment for Wireless (BREW). [#14562]

3. Compile SWIG-generated code with the -fno-strict-aliasing flag when using the GNU gcc compiler. [#14953]

4. Changed include files so ENOENT is resolved on Windows. [#15078]

5. Port Berkeley DB to WinCE. [#15312]

6. Port Berkeley DB to S60. [#15371]

7. Add the db_hotbackup executable to the Windows MSI installer. [#15372]

8. Change the db_hotbackup utility to use the Berkeley DB library portability layer. [#15415]

9. Re-write the GNU gcc mutex implementation on the x86 platform to avoid compiler errors. [#15461]

10. Fix a bug with non-HFS filesystems under OS X which could affect data durability. [#15501]

Chapter 44. Upgrading Berkeley DB 4.6 applications to Berkeley DB 4.7

Release 4.7: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.6 release interfaces to the Berkeley DB 4.7 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.7: Run-time configuration

In historic Berkeley DB releases, there were separate sleep and yield functions to be configured at run-time using the db_env_set_func_sleep and db_env_set_func_yield functions. These functions have been merged in the Berkeley DB 4.7 release. The replacement function should always yield the processor, and optionally wait for some period of time before allowing the thread to run again.

Applications using the Berkeley DB run-time configuration interfaces should merge the functionality of their sleep and yield functions into a single configuration function.

In the 4.7 Berkeley DB release, the db_env_set_func_map and db_env_set_func_unmap functions have been replaced. This change fixes problems where applications using the Berkeley DB run-time configuration interfaces could not open multiple DB_ENV class handles for the same database environment in a single application or join existing database environments from within multiple processes.

Applications wanting to replace the Berkeley DB region creation functionality should replace their db_env_set_func_map and db_env_set_func_unmap calls with a call to the db_env_set_func_region_map function. Applications wanting to replace the Berkeley DB region file mapping functionality should replace their db_env_set_func_map and db_env_set_func_unmap calls with a call to the db_env_set_func_file_map function.

Release 4.7: Replication API

The Berkeley DB base replication API DB_ENV->rep_elect(), DB_ENV->rep_get_nsites() DB_ENV->rep_set_nsites(), DB_ENV->rep_get_priority() and DB_ENV->rep_set_priority() methods now take arguments of type u_int32_t rather than int. Applications may need to change the types of arguments to these methods, or cast arguments to these methods to avoid compiler warnings.

Release 4.7: Tcl API

The Berkeley DB Tcl API does not attempt to avoid evaluating input as Tcl commands. For this reason, it may be dangerous to pass unreviewed user input through the Berkeley DB Tcl API, as the input may subsequently be evaluated as a Tcl command. To minimize the effectiveness of a Tcl injection attack, the Berkeley DB Tcl API in the 4.7 release routine resets process' effective user and group IDs to the real user and group IDs.

Release 4.7: DB_ENV->set_intermediate_dir

Historic releases of Berkeley DB contained an undocumented DB_ENV method named DB_ENV->set_intermediate_dir, which configured the creation of any intermediate directories needed during recovery. This method has been standardized as the DB_ENV->set_intermediate_dir_mode() method.

Applications using DB_ENV->set_intermediate_dir should be modified to use the DB_ENV->set_intermediate_dir_mode() method instead.

Release 4.7: Log configuration

In the Berkeley DB 4.7 release, the logging subsystem is configured using the DB_ENV->log_set_config() method instead of the previously used DB_ENV->set_flags() method.

The DB_ENV->set_flags() method no longer accepts the flags DB_DIRECT_LOG, DB_DSYNC_LOG, DB_LOG_INMEMORY or DB_LOG_AUTOREMOVE. Applications should be modified to use the equivalent flags accepted by the DB_ENV->log_set_config() method.

Previous DB_ENV->set_flags() flag	Replacement DB_ENV->log_set_config() flag
DB_DIRECT_LOG	DB_LOG_DIRECT
DB_DSYNC_LOG	DB_LOG_DSYNC
DB_LOG_INMEMORY	DB_LOG_IN_MEMORY
DB_LOG_AUTOREMOVE	DB_LOG_AUTO_REMOVE

Release 4.7: Upgrade Requirements

The log file format changed in the Berkeley DB 4.7 release.

No database formats changed in the Berkeley DB 4.7 release.

The Berkeley DB 4.7 release does not support live replication upgrade from the 4.2 or 4.3 releases, only from the 4.4 and later releases.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.7.25 Change Log

Database or Log File On-Disk Format Changes:

1. The log file format changed in 4.7.

New Features:

1. The lock manager may now be fully partitioned, improving performance on some multi-CPU systems. [#15880]

2. Replication groups are now architecture-neutral, supporting connections between differing architectures (big-endian or little-endian, independent of structure padding). [#15787] [#15840]

3. Java: A new Direct Persistence Layer adds a built-in Plain Old Java Object (POJO)-based persistent object model, which provides support for complex object models without compromises in performance. For an introduction to the Direct Persistence Layer API, see Getting Started with Data Storage. [#15936]

4. Add the DB_ENV->set_intermediate_dir_mode method to support the creation of intermediate directories needed during recovery. [#15097]

5. The DB_ENV->failchk method can now abort transactions for threads, which have failed while blocked on a concurrency lock. This significantly decreases the need for database environment recovery after thread of control failure. [#15626]

6. Replication Manager clients now can be configured to monitor the connection to the master using heartbeat messages, in order to promptly discover connection failures. [#15714]

7. The logging system may now be configured to pre-zero log files when they are created, improving performance on some systems. [#15758]

Database Environment Changes:

1. Restructure aborted page allocation handling on systems without an ftruncate system call. This enables the Berkeley DB High Availability product on systems, which do not support ftruncate. [#15602]

2. Fix a bug where closing a database handle after aborting a transaction which included a failed open of that handle could result in application failure. [#15650]

3. Fix minor memory leaks when closing a private database environment. [#15663]

4. Fix a bug leading to a panic of "unpinned page returned" if a cursor was used for a delete multiple times and deadlocked during one of the deletes. [#15944]

5. Optionally signal processes still running in the environment before running recovery. [#15984]

Concurrent Data Store Changes:

None.

General Access Method Changes:

1. Fix a bug where closing a database handle after aborting a transaction which included a failed open of that database handle could result in application failure. [#15650]

2. Fix a bug that could cause panic in a database environment configured with POSIX-style thread locking, if a database open failed. [#15662]

3. Fix bug in the DB->compact method which could cause a panic if a thread was about to release a page while another thread was truncating the database file. [#15671]

4. Fix an obscure case of interaction between a cursor scan and delete that was prematurely returning DB_NOTFOUND. [#15785]

5. Fix a bug in the DB->compact method where if read-uncommitted was configured, a reader reading uncommitted data my see an inconsistent entry between when the compact method detects an error and when it aborts the enclosing transaction. [#15856]

6. Fix a bug in the DB->compact method where a thread of control mail fail if two threads are compacting the same section of a Recno database. [#15856]

7. Fix a bug in DB->compact method, avoid an assertion failure when zero pages can be freed. [#15965]

8. Fix a bug return a non-zero error when DB->truncate is called with open cursors. [#15973]

9. Fix a bug add HANDLE_DEAD checking for DB cursors. [#15990]

10. Fix a bug to now generate errors when DB_SEQUENCE->stat is called without first opening the sequence. [#15995]

11. Fix a bug to no longer dereference a pointer into a hash structure, when hash functionality is disabled. [#16095]

Btree Access Method Changes:

None.

Hash Access Method Changes:

1. Fix a bug where a database store into a Hash database could self-deadlock in a database environment configured for the Berkeley DB Concurrent Data Store product, and with a free-threaded DB_ENV or DB handle. [#15718]

Queue Access Method Changes:

1. Fix a bug that could cause a put or delete of a queue element to return a DB_NOTGRANTED error, if blocked. [#15933]

Recno Access Method Changes:

1. Expose db_env_set_func_malloc, db_env_set_func_realloc, and db_env_set_func_free through the Windows API for the DB dll. [#16045]

C-specific API Changes:

None.

Java-specific API Changes:

1. Fix a bug where enabling MVCC on a database through the Java API was ignored. [#15644]

2. Fixed memory leak bugs in error message buffering in the Java API. [#15843]

3. Fix a bug where Java SecondaryConfig was not setting SecondaryMultiKeyCreator from the underlying db handle [OTN FORUM]

4. Fix a bug so that getStartupComplete will now return a boolean instead of an int. [#16067]

5. Fix a bug in the Java API, where Berkeley DB would hang on exit when using replication. [#16142]

Direct Persistence Layer (DPL), Bindings and Collections API:

1. A new Direct Persistence Layer adds a built-in Plain Old Java Object (POJO)-based persistent object model, which provides support for complex object models without compromises in performance. For an introduction to the Direct Persistence Layer API, see Getting Started with Data Storage. [#15936]

2. Fixed a bug in the remove method of the Iterator instances returned by the StoredCollection.iterator method in the collections package. This bug caused ArrayIndexOutOfBoundsException in some cases when calling next, previous, hasNext or hasPrevious after calling remove. (Note that this issue does not apply to StoredIterator instances returned by the StoredCollection.storedIterator method.) This bug was reported in this forum thread: http://forums.oracle.com/forums/thread.jspa?messageID=2187896 [#15858]

3. Fixed a bug in the remove method of the StoredIterator instances returned by StoredCollection.storedIterator method in the collections package. If the sequence of methods next-remove-previous was called, previous would sometimes return the removed record. If the sequence of methods previous-remove-next was called, next would sometimes return the removed record. (Note that this issue does not apply to Iterator instances returned by the StoredCollection.iterator method.) [#15909]

4. Fixed a bug that causes a memory leak for applications where many Environment objects are opened and closed and the CurrentTransaction or TransactionRunner class is used. The problem was reported in this JE Forum thread: http://forums.oracle.com/forums/thread.jspa?messageID=1782659 [#15444]

5. Added StoredContainer.areKeyRangesAllowed method. Key ranges and the methods in SortedMap and SortedSet such as subMap and subSet are now explicitly disallowed for RECNO and QUEUE databases -- they are only supported for BTREE databases. Before, using key ranges in a RECNO or QUEUE database did not work, but was not explicitly prohibited in the Collections API. [#15936]

Tcl-specific API Changes:

1. The Berkeley DB Tcl API does not attempt to avoid evaluating input as Tcl commands. For this reason, it may be dangerous to pass unreviewed user input through the Berkeley DB Tcl API, as the input may subsequently be evaluated as a Tcl command. To minimize the effectiveness of a Tcl injection attack, the Berkeley DB Tcl API in the 4.7 release routine resets process' effective user and group IDs to the real user and group IDs. [#15597]

RPC-specific Client/Server Changes:

None.

Replication Changes:

1. Fix a bug where a master failure resulted in multiple attempts to perform a "fast election"; subsequent elections, when necessary, now use the normal nsites value. [#15099]

2. Replication performance enhancements to speed up failover. [#15490]

3. Fix a bug where replication could self-block in a database environment configured for in-memory logging. [#15503]

4. Fix a bug where replication would attempt to read log file version numbers in a database configured for in-memory logging. [#15503]

5. Fix a bug where log files were not removed during client initialization in a database configured for in-memory logging. [#15503]

6. The 4.7 release no longer supports live replication upgrade from the 4.2 or 4.3 releases, only from the 4.4 and later releases. [#15602]

7. Fix a bug where replication could re-request missing records on every arriving record. [#15629]

8. Change the DB_ENV->rep_set_request method to use time, not the number of messages, when re-requesting missed messages on a replication client. [#15629]

9. Fix a minor memory leak on the master when updating a client during internal initialization. [#15634]

10. Fix a bug where a client error when syncing with a new replication group master could result in an inability to ever re-join the group. [#15648]

11. Change dbenv->rep_set_request to use time-based values instead of counters. [#15682]

12. Fix a bug where a LOCK_NOTGRANTED error could be returned from the DB_ENV->rep_process_message method, instead of being handled internally by replication. [#15685]

13. Fix a bug where the Replication Manager would reject a fresh connection from a remote site that had crashed and restarted, displaying the message: "redundant incoming connection will be ignored". [#15731]

14. The Replication Manager now supports dynamic negotiation of the best available wire protocol version, on a per-connection basis. [#15783]

15. Fix a bug, which could lead to slow performance of internal initialization under the Replication Manager, as evidenced by "queue limit exceeded" messages in verbose replication diagnostic output. [#15788]

16. Fix a bug where replication control message were not portable between replication clients with different endian architectures. [#15793]

17. Add a configuration option to turn off Replication Manager's special handling of elections in 2-site groups. [#15873]

18. Fix a bug making it impossible to call replicationManagerAddRemoteSite in the Java API after having called replicationManagerStart. [#15875]

19. Fix a bug where the DB_EVENT_REP_STARTUPDONE event could be triggered too early. [#15887]

20. Fix a bug where the rcvd_ts timestamp is reset when the user just changes the threshold. [#15895]

21. Fix a bug where the master in a 2-site replication group might wait for client acknowledgement, even when there was no client connected. [#15927]

22. Fix a bug, clean up and restart internal init if master log is gone. [#16006]

23. Fix a bug, ignore page messages that are from an old internal init. [#16075] [#16059]

24. Fix a bug where checkpoint records do not indicate a database was a named in-memory database. [#16076]

25. Fix a bug with in-memory replication, where we returned with the log region mutex held in an error path, leading to self-deadlock. [#16088]

26. Fix a bug which causes the DB_REP_CHECKPOINT_DELAY setting in rep_set_timeout() to be interpreted in seconds, rather than microseconds. [#16153]

XA Resource Manager Changes:

1. Fix a bug where the DB_ENV->failchk method and replication in general could fail in database environments configured for XA. [#15654]

Locking Subsystem Changes:

1. Fix a bug causing a lock or transaction timeout to not be set properly after the first timeout triggers on a particular lock id. [#15847]

2. Fix a bug that would cause a trap if DB_ENV->lock_id_free was passed an invalid locker id. [#16005]

3. Fix a bug when thread tracking is enabled where an attempt is made to release a mutex that is not lock. [#16011]

Logging Subsystem Changes:

1. Fix a bug, handle zero-length log records doing HA sync with in-memory logs. [#15838]

2. Fix a bug that could cause DB_ENV->failcheck to leak log region memory. [#15925]

3. Fix a bug where the abort of a transaction that opened a database could leak log region memory. [#15953]

4. Fix a bug that could leak memory in the DB_ENV->log_archive interface if a log file was not found. [#16013]

Memory Pool Subsystem Changes:

1. Fix multiple MVCC bugs including a race, which could *result in incorrect data being returned* to the application. [#15653]

2. Fixed a bug that left an active file in the buffer pool after a database create was aborted. [#15918]

3. Fix a bug where there could be uneven distribution of pages if a single database and multiple cache regions are configured. [#16015]

4. Fix a bug where DB_MPOOLFILE->set_maxsize was dropping the wrong mutex after open. [#16050]

Mutex Subsystem Changes:

1. Fix a bug where mutex contention in database environments configured for hybrid mutex support could result in performance degradation. [#15646]

2. Set the DB_MUTEX_PROCESS_ONLY flag on all mutexes in private environments, they can't be shared and so we can use the faster, intra-process only mutex implementations [#16025]

3. Fix a bug so that mutexes are now removed from the environment signature if mutexes are disabled. [#16042]

Transaction Subsystem Changes:

1. Fix a bug that could cause a checkpoint to selfblock attempting to flush a file, when the file handle was closed by another thread during the flush. [#15692]

2. Fix a bug that could cause DB_ENV->failcheck to hang if there were pending prepared transactions in the environment. [#15925]

3. Prepared transactions will now use the sync setting from the environment. Default to flushing the log on commit (was nosync). [#15995]

4. If __txn_getactive fails, we now return with the log region mutex held. This is not a bus since __txn_getactive cannot really fail. [#16088]

Utility Changes:

1. Update db_stat with -x option for mutex stats

2. Fix an incorrect assumption about buffer size when getting an overflow page in db_verify. [#16064]

Configuration, Documentation, Sample Application, Portability and Build Changes:

1. Fix an installation bug where the Berkeley DB PHP header file was not installed in the correct place.
2. Merge the run-time configuration sleep and yield functions. [#15037]
3. Fix Handle_DEAD and other expected replication errors in the C++ sample application ReqQuoteExample.cpp. [15568]
4. Add support for monotonic timers. [#15670]
5. Fix bugs where applications using the db_env_func_map and db_env_func_unmap run-time configuration functions could not join existing database environments, or open multiple DB_ENV handles for a single environment. [#15930]
6. Add documentation about building Berkeley DB for VxWorks 6.x.
7. Remove the HAVE_FINE_GRAINED_LOCK_MANAGER flag, it is obsolete in 4.7.
8. Fix a bug in ex_rep, add a missing break which could cause a segment fault.
9. Fix build warnings from 64 bit Windows build. [#16029]
10. Fix an alignment bug on ARM Linux. Force the assignment to use memcpy. [#16125]
11. Fix a bug in the Windows specific code of ex_sequence.c, where there was an invalide printf specifier. [#16131]
12. Improve the timer in ex_tpcb to use high resolution timers. [#16154]
13. Mention in the documentation that env->open() requires DB_THREAD to be specified when using repmgr. [#16163]
14. Disable support for mmap on Windows CE. The only affect is that we do not attempt to mmap small read only databases into the mpool. [#16169]

Chapter 45. Upgrading Berkeley DB 4.7 applications to Berkeley DB 4.8

Release 4.8: Introduction

The following pages describe how to upgrade applications coded against the Berkeley DB 4.7 release interfaces to the Berkeley DB 4.8 release interfaces. This information does not describe how to upgrade Berkeley DB 1.85 release applications.

Release 4.8: Registering DPL Secondary Keys

Entity subclasses that define secondary keys must now be registered prior to storing an instance of the class. This can be done in two ways:

- The `EntityModel.registerClass()` method may be called to register the subclass before opening the entity store.
- The `EntityStore.getSubclassIndex()` method may be called to implicitly register the subclass after opening the entity store.

Failure to register the entity subclass will result in an `IllegalArgumentException` the first time an attempt is made to store an instance of the subclass. An exception will not occur if instances of the subclass have previously been stored, which allows existing applications to run unmodified in most cases.

This behavioral change was made to increase reliability. In several cases, registering an entity subclass has been necessary as a workaround. The requirement to register the subclass will ensure that such errors do not occur in deployed applications.

Release 4.8: Minor Change in Behavior of DB_MPOOLFILE->get

DB 4.8 introduces some performance enhancements, based on the use of shared/exclusive latches instead of locks in some areas of the internal buffer management code. This change will affect how the `DB_MPOOL` interface handles dirty buffers.

Because of these changes, `DB_MPOOLFILE->get` will now acquire an exclusive latch on the buffer if the `DB_MPOOL_DIRTY` or `DB_MPOOL_EDIT` flags are specified. This could lead to an application deadlock if the application tries to fetch the buffer again, without an intervening `DB_MPOOLFILE->put` call.

If your application uses the `DB_MPOOL` interface, and especially the `DB_MPOOL_DIRTY` and `DB_MPOOL_EDIT` flags, you should review your code to ensure that this behavior change does not cause your application to deadlock.

Release 4.8: Dropped Support for fcntl System Calls

Berkeley DB no longer supports mutex implementations based on the `fcntl` system call. If you have been configuring Berkeley DB to use this type of mutex, you need to either switch to a different mutex type or contact the Berkeley DB team for support.

Release 4.8: Upgrade Requirements

The log file format changed in the Berkeley DB 4.8 release.

No database formats changed in the Berkeley DB 4.8 release.

The Berkeley DB 4.8 release does not support live replication upgrade from the 4.2 or 4.3 releases, only from the 4.4 and later releases.

For further information on upgrading Berkeley DB installations, see Upgrading Berkeley DB installations (page 332).

Berkeley DB 4.8.30 Change Log

Changes between 4.8.26 and 4.8.30:

1. Limit the size of a log record generated by freeing pages from a database so it fits in the log file size. [#17313]
2. Fixed a bug that could cause a file to be removed if it was both the source and target of two renames within a transaction. [#18069]
3. Modified how we go about selecting a usable buffer in the cache. Place more emphasis on single version and obsolete buffers. [#18114]
4. Fixed a bug that could lead to btree structure corruption if the db->compact method runs out of locks. [#18361]
5. Allow any file to be truncated even if its not a db file. [#18373]
6. Avoid a segmentation fault error if the lock manager runs out of locks. [#18428]
7. Add dbreg close records for open but missing databases during recovery. [#18459]
8. Fixed a bug which could occur when using bulk transfer with Replication Manager. Replication Manager may send messages even though its connections have already been closed, leading in rare circumstances to spurious EBADF error reports, or possibly even arbitrary memory corruption. [#18469]
9. Fixed a bug in C# API where db.HasMultiple was throwing an incorrect exception. [#18483]
10. Fixed a bug where populating a SecondaryDatabase on open could lead to an OutOfMemoryException.. [#18529]

11. Fixed a bug where entries in the db register file did not get cleared out properly after recovery takes place. This will permit a process to perform a dbenv->close and then reconnect to environment without needing to stop the process. [#18535]

Known bugs in 4.8

1. Sharing logs across mixed-endian systems does not work.[#18032]

Changes between 4.8.24 and 4.8.26:

1. Fixed a bug where the truncate log record could be too large when freeing too many pages during a compact. [#17313]

2. Fixed a bug where the deadlock detector might not run properly. [#17555]

3. Fixed three bugs related to properly detecting thread local storage for DbStl. [#17609] [#18001] [#18038]

4. Fixed a bug that prevented some of our example code from running correctly in a Windows environment. [#17627]

5. Fixed a bug where a "unable to allocate space from buffer cache" error was improperly generated. [#17630]

6. Fixed a bug where DB->exists() did not accept the DB_AUTO_COMMIT flag. [#17687]

7. Fixed a bug where DB_TXN_SNAPSHOT was not getting ignored when DB_MULTIVERSION not set. [#17706]

8. Fixed a bug that prevented callback based partitioning through the Java API. [#17735]

9. Fixed a replication bug where log files were not automatically removed from the client side. [#17899]

10. Fixed a bug where code generated from db_sql stored the key in both the data and key DBTs. [#17925]

11. Fixed a bug that prevented a sequence from closing properly after the EntityStore closed. [#17951]

12. Fixed a bug where gets fail if the DB_GET_BOTH_FLAG is specified in a hash, sorted duplicates database.[#17997]

Changes between 4.8.21 and 4.8.24:

1. Fixed a bug in the C# API where applications in a 64-bit environment could hang. [#17461]

2. Fixed a bug in MVCC where an exclusive latch was not removed when we couldn't obtain a buffer. [#17479]

3. Fixed a bug where a lock wasn't removed on a non-transactional locker. [#17509]

4. Fixed a bug which could trigger an assertion when performing a B-tree page split and running out of log space or with MVCC enabled. [#17531]

5. Fixed a bug in the repquote example that could cause the application to crash. [#17547]

6. Fixed a couple of bugs when using the GCC 4.4 compiler to build the examples and the dbstl API. [#17504] [#17476]

7. Fixed an incorrect representation of log system configuration info. [#17532]

Changes between 4.7 and 4.8.21:

Database or Log File On-Disk Format Changes:

1. The log file format changed in 4.8.

New Features:

1. Improved scalability and throughput when using BTree databases especially when running with multiple threads that equal or exceed the number of available CPUs.

2. Berkeley DB has added support for C#. In addition to the new C# api, C# specific tests and sample applications were also added. [#16137]

3. Berkeley DB has added an STL API, which is compatible with and very similar to C++ Standard Template Library (STL). Tests and sample applications and documentation were also added. [#16217]

4. Berkeley DB has added database partitioning. BTree or Hash databases may now be partitioned across multiple directories. Partitioned databases can be used to increase concurrency and to improve performance by spreading access across disk subsystems. [#15862]

5. Berkeley DB now supports bulk insertion and deletion of data. Similar to the bulk get interface, the bulk put and bulk delete allow the developer to populate a buffer of key-value pairs and then pass it to the BDB library with a single API call.

6. Berkeley DB now supports compression when using BTree.

7. Berkeley DB introduces a new utility named db_sql which replaces db_codegen. Similar to db_codegen, db_sql accepts an input file with DDL statements and generates a Berkeley DB application using the C API that creates and performs CRUD operations on the defined tables. The developer can then use that code as a basis for further application development.

8. The Replication Manager now supports shared access to the Master database environment from multiple processes. In earlier versions, multiple process support on the Master required use of the Base Replication API. [#15982]

9. Foreign Key Support has been added to Berkeley DB.

10. Several enhancements were made to DB_REGISTER & DB_ENV->failchk().

11. Berkeley now supports 100% in-memory replication.

12. Berkeley DB now has the ability to compare two cursors for equality. [#16811]

Database Environment Changes:

1. Fixed a bug that could cause an allocation error while trying to allocate thread tracking information for the DB_ENV->failcheck system. [#16300]

2. Fixed a bug that could cause a trap if an environment open failed and failchk thread tracking was enabled.[#16770]

Concurrent Data Store Changes:

None.

General Access Method Changes:

1. Fixed a bug where doing an insert with secondary indices and the NOOVERWRITE flag could corrupt the secondary index. [#15912]

2. Fixed a possible file handle leak that occurred while aborting the create of a database whose metadata page was not initialized. [#16359]

3. Fixed a bug so that we now realloc the filename buffer only if we need it to grow. [#16385] [#16219]

4. Fixed a race freeing a transaction object when using MVCC. [#16381]

5. Added missing get methods for the DB and DB_ENV classes where there already was a corresponding set method. [#16505]

6. Fixed a bug to now ensure that DB_STAT_SUBSYSTEM is distinct from other stat flags. [#16798]

7. Fixed a bug related to updating multiple secondary keys (using DB_MULTIPLE). [#16885]

8. Fixed a bug so that verify (db->verify, db_verify) will now report when it cannot read a page rather than just saying the database is bad. [#16916]

9. Fixed a bug that could cause memory corruption if a transaction allocating a page aborted while DB->compact was running on that database. [#16862]

10. Fixed a bug where logging was occurring during remove of an in-memory database when the DB_TXN_NOT_DURABLE flag was set. [#16571]

11. Fixed a bug to remove a race condition during database/file create. [#17020]

12. Fixed a bug where a call to DB->verify and specifying DB_SALVAGE could leak memory when the call returned. [#17161]

13. Fixed a bug to avoid accessing freed memory during puts on primaries with custom comparators. [#17189]

14. Fixed a bug that could cause old versions of pages to be written over new versions if an existing database is opened with the DB_TRUNCATE flag. [#17191]

Btree Access Method Changes:

1. Fixed a bug which could cause DB->compact to fail with DB_NOTFOUND or DB_PAGE_NOTFOUND if the height of the tree was reduced by another thread while compact was active. The bug could also cause a page split to trigger splitting of internal nodes which did not need to be split. [#16192]

2. Fixed a bug that caused Db->compact to loop if run on an empty RECNO database when there were pages in the free list. [#16778]

3. Added a new flag, DB_OVERWRITE_DUP, to DB->put and DBC->put. This flag is equivalent to DB_KEYLAST in almost all cases: the exception is that with sorted duplicates, if a matching key/data pair exists, we overwrite it rather than returning DB_KEYEXIST. [#16803]

Hash Access Method Changes:

1. Fixed a bug to now force a group allocation that rolls forward to reinit all the pages. Otherwise a previous aborted allocation may change the header. [#15414]

2. Fixed a bug to now return the expected buffer size on a DB_BUFFER_SMALL condition. [#16881]

Queue Access Method Changes:

1. Fixed a bug that would cause the LSN reset functionality to not process queue extents. [#16213]

2. Fixed a bug that prevented a partial put on a queue database with secondaries configured. [#16460]

3. Fixed a bug to now prevent an unpinned page to be returned if a delete from a HASH database deadlocked. [#16371]

4. Fixed a bug that could cause a queue extent to be recreated if an application deleted a record that was already deleted in that extent. [#17004]

5. Added the DB_CONSUME flag to DB->del and DBC->del to force adjustment of the head of the queue. [#17004]

Recno Access Method Changes:

1. Fixed a bug which could cause DB->compact of a RECNO database to loop if the number of pages on the free list was reduced by another thread while compact was active. [#16199]

2. Fixed a bug that occurs when deleting from a Recno database and using DB_READ_UNCOMMITTED where we could try to downgrade a lock twice. [#16347]

3. Fixed a bug to now disallow passing DB_DUP and DB_RECNUM together to __db_set_flags. [#16585]

C-specific API Changes:

1. Add get functions for each set functions of DB and DB_ENV structures which didn't have one.[#16505]

C++-specific API Changes:

1. The get and set_lk_partitions methods are now available.

2. Add get functions for each set functions of Db and DbEnv classes which didn't have one.[#16505]

3. Fixed a memory leak when using nested transactions.[#16956]

Java-specific API Changes:

1. Fixed a bug where the replication finer-grained verbose flags were not available in the Java API. [#15419]

2. Fixed a bug in the BTree prefix compression API when called from the Java API. DBTs were not properly initialized. [#16417]

3. Fixed a bug so that LogCursor will work correctly from the Java API. [#16827]

4. Fixed a bug so that position(), limit() and capacity() of ByteBuffers are obeyed by DatabaseEntry objects. [#16982]

Direct Persistence Layer (DPL), Bindings and Collections API:

1. The StoredMap class now implements the standard java.util.concurrent.ConcurrentMap interface. [#15382]

2. Report a meaningful IllegalArgumentException when @Persistent is incorrectly declared on an enum class. Before, the confusing message Persistent class has non-persistent superclass: java.lang.Enum was reported. [#15623]

3. Report a meaningful IllegalArgumentException when @Persistent is incorrectly declared on an interface. Before, a NullPointerException was reported. [#15841]

4. Several validation checks have been added or corrected having to do with entity subclasses, which are @Persistent classes that extend an @Entity class. [#16077]

5. Optimized marshaling for large numbers of embedded objects improving performance. [#16198]

6. The StoredMap class now implements the Java 1.5 ConcurrentMap interface. [#16218]

7. Fix a DPL bug that caused exceptions when using a class Converter for an instance containing non-simple fields. [#16233]

8. Add EntityCursor.setCacheMode and getCacheMode. See the com.sleepycat.je.CacheMode class for more information. [#16239]

9. Fix a bug that prevents evolution of @SecondaryKey information in an entity subclass (a class that extends an @Entity class). [#16253]

10. Report a meaningful IllegalArgumentException when @Persistent or @Entity is incorrectly used on an inner class (a non-static nested class). Before, the confusing message No default constructor was reported. [#16279]

11. Improved the reliability of Entity subclasses that define secondary keys by requiring that they be registered prior to storing an instance of the class. [#16399]

12. Fix a bug that under certain circumstances causes "IllegalArgumentException: Not a key class" when calling EntityStore.getSubclassIndex, EntityStore.getPrimaryConfig, EntityStore.getSecondaryConfig, or PrimaryIndex.put, and a composite key class is used. [#16407]

13. Fixed a bug so that one can now compile DPL in the Java API on Windows. [#16570]

14. The com.sleepycat.collections.TransactionRunner.handleException method has been added to allow overriding the default transaction retry policy. See the javadoc for this method for more information. [#16574]

15. Fix a bug that causes an assertion to fire or a NullPointerException (when assertions are disabled) from the EntityStore constructor. The problem occurs only when the previously created EntityStore contains an entity with a secondary key definition in which the key name has been overridden and is different than the field name. [#16819]

16. Key cursors have been optimized to significantly reduce I/O when the READ_UNCOMMITTED isolation mode is used. See EntityIndex.keys for more information. [#16859]

17. Report a meaningful IllegalArgumentException when NULLIFY is used with a @SecondaryKey and the field is a primitive type. Before, the confusing message Key field object may not be null was reported. [#17011]

18. Enum fields may now be used as DPL keys, including primary keys, secondary keys, and fields of composite key classes. Comparators are supported for composite key classes containing enum fields. [#17140]

19. Fix a bug that prevented the use of custom key comparisons (composite key classes that implement Comparable) for secondary keys defined as ONE_TO_MANY or MANY_TO_MANY.[#17207]

20. The db.jar file now contains a Premain class which enables bytecode enhancement using the JVM instrumentation commands. The built-in proxy classes are also now enhanced in

the db.jar file, which enables off-line bytecode enhancement. For more information on DPL bytecode enhancement and how to use both instrumentation and off-line enhancement, please see the com.sleepycat.persist.model.ClassEnhancer javadoc. [#17233]

Tcl-specific API Changes:

1. The mutex API is now available when using Tcl. [#16342]

RPC-specific Client/Server Changes:

- RPC support has been removed from Berkeley DB. [#16785]

Replication Changes:

1. Improved testing of initial conditions for rep and repmgr APIs and added heartbeat timeouts to rep_get_timeout.[#14977]

2. Added DB_REP_CONF_INMEM replication configuration flag to store replication information exclusively in-memory without creating any files on-disk. [#15257]

3. Added repmgr support for multi-process shared env [#15982]

4. Fixed a bug where opening a cursor from a database handle failed to check whether the database handle was still fresh. If the database handle had been invalidated by a replication client synchronizing with a new master, it could point to invalid information. [#15990]

5. Fixed a bug so that if LOG_REQ gets an archived LSN, replication sends VERIFY_FAIL. [#16004]

6. Added timestamp and process/thread id to replication verbose messages. [#16098]

7. Fixed a bug where, in very rare circumstances, two repmgr sites could connect to each other at the exact same time, the connection attempts "collide" and fail, and the same collision repeats in time synchronization indefinitely. [#16114]

8. Fixed a bug where a missing database file (FILE_FAIL error condition) can interrupt a client synchronization without restarting it. [#16130]

9. Fixed a bug by adding REP_F_INREPSTART flag to prevent racing threads in rep_start. [#16247]

10. Fixed a bug to not return HOLDELECTION if we are already in the middle of an election. Updated the egen so the election thread will notice. [#16270]

11. Fixed a bug in buffer space computation, which could have led to memory corruption in rare circumstances, when using bulk transfer. [#16357]

12. Fixed a bug that prevented replication clients from opening a sequence. The sequence is opened for read operations only. [#16406]

13. Fixed a bug by removing an assertion about priority in elections. It is not correct because it could have changed by then. Remove unused recover_gen field. [#16412]

14. Fixed a bug to now ignore a message from client if it is an LSN not recognized in a LOG_REQ. [#16444]

15. Fixed a bug so that on POSIX systems, repmgr no longer restores default SIGPIPE action upon env close, if it was necessary to change it during start-up. This allows remaining repmgr environments within the same process, if any, to continue operating after one of them is closed. [#16454]

16. After a replication client restarts with recovery, any named in-memory databases are now re-materialized from the rest of the replication group upon synchronization with the master. [#16495]

17. Fixed a bug by adding missing rep_get_config flags. [#16527]

18. Instead of sleeping if the bulk buffer is in transmission, return so that we can send as a singleton. [#16537]

19. Fixed a bug by changing __env_refresh to not hit assert on -private -rep env with an in-memory database. [#16546]

20. Fixed a bug in the Windows implementation of repmgr where a large number of commit threads concurrently awaiting acknowledgments could result in memory corruption, and leaking Win32 Event Objects. [#16548]

21. Fixed a bug by changing repmgr to count a dropped connection when noticing a lacking heartbeat; fixed hearbeat test to check for election, rather than connection drop count, and more reasonable time limit; fixed test to poll until desired result, rather than always sleeping max possible time. [#16550]

22. Fixed "master changes" stat to count when local site becomes master too. [#16562]

23. Fixed a bug where a c2c client would send UPDATE_REQ to another client [#16592]

24. Removed code to proactively expire leases when we don't get acks. Leases maintain their own LSNs to know. [#16494]

25. Fixed a bug where a client may not sync pages during internal init. [#16671]

26. Fixed a bug where a client that received and skipped a log record from the master during an election, then won the election, could then try to request a copy of the skipped log record. The result was an attempt to send a request to the local site, which is invalid: this could confuse a replication Base API application, or cause the Replication Manager to crash. [#16700]

27. Fixed a bug which could have caused data loss or corruption (at the client only) if a replication client rolled back existing transactions in order to synchronize with a new master, and then crashed/recovered before a subsequent checkpoint operation had been replicated from the master. [#16732]

28. Fixed a bug so that replication now retries on DB_LOCK_NOTGRANTED. [#16741]

29. Fixed a potential deadlock in rep_verify_fail. [#16779]

30. Fixed a bug so that an application will no longer segv if nsites given was smaller than number of sites that actually exists. [#16825]

XA Resource Manager Changes:

1. The XA Resource Manager has been removed from Berkeley DB. [#6459]

Locking Subsystem Changes:

1. Fixed a bug to prevent unlocking a mutex twice if we ran out of transactional locks. [#16285]

2. Fixed a bug to prevent a segmentation trap in __lock_open if there were an error during the opening of an environment. [#16307]

3. Fixed a bug to now avoid a deadlock if user defined locks are used only one lock partition is defined.[#16415]

4. Fixed concurrency problems in __dd_build, __dd_abort by adding LOCK_SYSTEM_LOCK() calls to __dd_build and __dd_abort. [16489]

5. Fixed a bug that could cause a panic if a transaction which updated a database that was supporting READ_UNCOMMITED readers aborted and it hit a race with a thread running the deadlock detector. [#16490]

6. Fixed a race condition in deadlock detection that could overwrite heap. [#16541]

7. Fixed a bug so that DB_STAT_CLEAR now restores the value of st_partitions. [#16701]

Logging Subsystem Changes:

1. Fixed a bug so that the header checksum is only ignored when the log is from a previous version [#16281]

2. Fixed a bug by removing a possible race condition with logc_get(DB_FIRST) and log archiving. [#16387]

3. Fixed a bug that could cause a recovery failure of a create of a database that was aborted. [#16824]

4. An in-memory database creation has an intermediate phase where we have a semi-open DBP. If we crash in that state, then recovery was failing because it tried to use a partically open database handle. This fix checks for that case, and avoids trying to undo page writes for databases in that interim step. [#17203]

Memory Pool Subsystem Changes:

1. Fixed a bug that occurred after all open handles on a file are closed. Needed to clear the TXN_NOT_DURABLE flag (if set) and mark the file as DURABLE_UNKNOWN in the memory pool. [#16091]

2. Fixed a possible race condition between dirtying and freeing a buffer that could result in a panic or corruption. [#16530]

3. Fixed a memory leak where allocated space for temporary file names are not released. [#16956]

Mutex Subsystem Changes:

1. Fixed a bug when using mutexes for SMP MIPS/Linux systems. [#15914]

2. POSIX mutexes are now the default on Solaris. [#16066]

3. Fixed a bug in mutex allocation with multiple cache regions. [#16178]

4. Fixed MIPS/Linux mutexes in 4.7. [#16209]

5. Fixed a bug that would cause a mutex to be unlocked a second time if we ran out of space while tracking pinned pages. [#16228]

6. Fixed a bug Sparc/GCC when using test-and-set mutexes. They are now aligned on an 8-byte boundary. [#16243]

7. Fixed a bug to now prevent a thread calling DB_ENV->failcheck to hang on a mutex held by a dead thread. [#16446]

8. Fixed a bug so that __db_pthread_mutex_unlock() now handles the failchk case of finding a busy mutex which was owned by a now-dead process. [#16557]

9. Removed support for the mutex implementation based on the "fcntl" system call. Anyone configuring Berkeley DB to use this type of mutex in an earlier release will need to either switch to a different mutex type or contact Oracle for support. [#17470]

Test Suite Changes

1. Fixed a bug when using failchk(), where a mutex was not released. [#15982]

2. Added a set of basic repmgr tests to run_std and run_all. [#16092]

3. Added control wrapper for db_reptest to test suite. [#16161]

4. Fixed a bug to now skip tests if db_reptest is not configured. [#16161]

5. Changed name of run_db_in_mem to run_inmem_db, and run_inmem to run_inmem_log and made the arg orders consistent. [#16358]

6. Fixed a bug to now clean up stray handles when rep_verify doesn't work. [#16390]

7. Fixed a bug to avoid db_reptest passing the wrong flag to repmgr_start when there is already a master. [#16475]

8. Added new tests for abbreviated internal init. Fixed test not to expect in-memory database to survive recovery. [#16495]

9. Fix a bug, to add page size for txn014 if the default page size is too small. Move files instead of renaming directory for env015 on QNX. [#16627]

10. Added new rep088 test for log truncation integrity. [#16732]

11. Fixed a bug by adding a checkpoint in rep061 to make sure we have messages to process. Otherwise we could hang with client stuck in internal init, and no incoming messages to trigger rerequest. [#16781]

Transaction Subsystem Changes:

1. Fixed a bug to no longer generate an error if DB_ENV->set_flags (DB_TXN_NOSYNC) was called after the environment was opened. [#16492]

2. Fixed a bug to remove a potential hang condition in replication os_yield loops when DB_REGISTER used with replication by adding PANIC_CHECKS. [#16502]

3. Fix a bug to now release mutex obtained before special condition returns in __db_cursor_int and __txn_record_fname. [#16665]

4. Fixed a leak in the transaction region when a snapshot update transaction accesses more than 4 databases. [#16734]

5. Enabled setting of set_thread_count via the DB_CONFIG file. [#16878]

6. Fixed a mutex leak in some corner cases. [#16665]

Utility Changes:

1. The db_stat utility with the -RA flags will now print a list of known remote replication flags when using repmgr. [#15484]

2. Restructured DB salvage to walk known leaf pages prior to looping over all db pages. [#16219]

3. Fixed a problem with upgrades to 4.7 on big endian machines. [#16411]

4. Fixed a bug so that now db_load consistently returns >1 on failure. [#16765]

5. The db_dump utility now accepts a "-m" flag to dump information from a named in-memory database. [#16896]

6. Fixed a bug that would cause db_hotbackup to fail if a database file was removed while it was running. [#17234]

Configuration, Documentation, Sample Application, Portability and Build Changes:

1. Fixed a bug to now use the correct Perl include path. [#16058]

2. Updated the version of the Microsoft runtime libraries shipped. [#16058]

3. Upgraded the Visual Studio build files to be based on Visual Studio 8 (2005+). The build is now simplified. Users can still upgrade the Visual Studio 6.0 project files, if they want to use Visual Studio .NET (7.1) [#16108]

4. Expanded the ex_rep example with checkpoint and log archive threads, deadlock detection, new options for acknowledgment policy and bulk transfer, and use of additional replication features and events. [#16109]

5. Fixed a bug so that optimizations on AIX are re-enabled, avoiding incorrect code generation. [#16141]

6. Removed a few compiler warnings and three type redefinitons when using vxworks and the GNU compiler. [#16341]

7. Fixed a bug on Sparc v9 so that MUTEX_MEMBAR() now uses membar_enter() to get a #storeload barrier rather than just stbar's #storestor. [#16468]

8. Berkeley DB no longer supports Win9X and Windows Me (Millenium edition).

9. Fixed lock_get and lock_vec examples from the Java (and C#) API. Updated the Java lock example. [#16506]

10. Fixed a bug to correctly handle the TPC-B history record on 64-bit systems. [#16709]

11. Add STL API to Linux build. Can be enabled via the --enable-stl flag. [#16786]

12. Add STL API to Windows build, by building the db_stl project in the solution. There are also stl's test and examples projects in this solution. [#16786]

13. Add support to build dll projects for WinCE, in order to enable users to build DB into a dll in addition to a static library.[#16625]

14. Fixed a weakness where several malloc/realloc return values are not checked before use.[#16664]

15. Enabled DB->compact for WinCE.[#15897]

16. HP-UX 10 is no longer supported.

Chapter 46. Test Suite

Running the test suite

Once you have started tclsh and have loaded the test.tcl source file (see Running the test suite under UNIX (page 298) and Running the test suite under Windows (page 318) for more information), you are ready to run the test suite. At the tclsh prompt, to run the standard test suite, enter the following:

```
% run_std
```

A more exhaustive version of the test suite runs all the tests several more times, testing encryption, replication, and different page sizes. After you have a clean run for run_std, you may choose to run this lengthier set of tests. At the tclsh prompt, enter:

```
% run_all
```

Running the standard tests can take from several hours to a few days to complete, depending on your hardware, and running all the tests will take at least twice as long. For this reason, the output from these commands are redirected to a file in the current directory named `ALL.OUT`. Periodically, a line will be written to the standard output, indicating what test is being run. When the test suite has finished, a final message will be written indicating the test suite has completed successfully or that it has failed. If the run failed, you should review the `ALL.OUT` file to determine which tests failed. Errors will appear in that file as output lines, beginning with the string "FAIL".

Tests are run in the directory `TESTDIR`, by default. However, the test files are often large, and you should use a filesystem with at least several hundred megabytes of free space. To use a different directory for the test directory, edit the file include.tcl in your build directory, and change the following line to a more appropriate value for your system:

```
set testdir ./TESTDIR
```

For example, you might change it to the following:

```
set testdir /var/tmp/db.test
```

Alternatively, you can create a symbolic link named TESTDIR in your build directory to an appropriate location for running the tests. Regardless of where you run the tests, the TESTDIR directory should be on a local filesystem. Using a remote filesystem (for example, an NFS mounted filesystem) will almost certainly cause spurious test failures.

Test suite FAQ

1. **The test suite has been running for over a day. What's wrong?**

   The test suite can take anywhere from some number of hours to several days to run, depending on your hardware configuration. As long as the run is making forward progress and new lines are being written to the `ALL.OUT` files, everything is probably fine.

Chapter 47. Distribution

Porting Berkeley DB to new architectures

Berkeley DB is generally easy to port to new architectures. Berkeley DB was designed to be as portable as possible, and has been ported to a wide variety of systems, from Wind River's Tornado system, to VMS, to Windows/NT and Windows/95, and most existing UNIX platforms. It runs on 16, 32 and 64-bit machines, little or big-endian. The difficulty of a port depends on how much of the ANSI C and POSIX 1003.1 standards the new architecture offers.

An abstraction layer separates the main Berkeley DB code from the operating system and architecture specific components. This layer is comprised of approximately 2500 lines of C language code, found in the `os` subdirectory of the Berkeley DB distribution. The following list of files include functionality that may need to be modified or implemented in order to support a new architecture. Within each file, there is usually one, but sometimes several functions (for example, the `os_alloc.c` file contains the malloc, calloc, realloc, free, and strdup functions).

Source file	Description
os_abs.c	Return if a filename is an absolute pathname
os_alloc.c	ANSI C malloc, calloc, realloc, strdup, free front-ends
os_clock.c	Return the current time-of-day
os_config.c	Return run-time configuration information
os_dir.c	Read the filenames from a directory
os_errno.c	Set/get the ANSI C errno value
os_fid.c	Create a unique ID for a file
os_fsync.c	POSIX 1003.1 fsync front-end
os_handle.c	Open file handles
os_id.c	Return thread ID
os_map.c	Map a shared memory area
os_method.c	Run-time replacement of system calls
os_oflags.c	Convert POSIX 1003.1 open flags, modes to Berkeley DB flags
os_open.c	Open file handles
os_region.c	Map a shared memory area
os_rename.c	POSIX 1003.1 rename call
os_root.c	Return if application has special permissions
os_rpath.c	Return last pathname separator
os_rw.c	POSIX 1003.1 read/write calls
os_seek.c	POSIX 1003.1 seek call

Source file	Description
os_sleep.c	Cause a thread of control to release the CPU
os_spin.c	Return the times to spin while waiting for a mutex
os_stat.c	POSIX 1003.1 stat call
os_tmpdir.c	Set the path for temporary files
os_unlink.c	POSIX 1003.1 unlink call

All but a few of these files contain relatively trivial pieces of code. Typically, there is only a single version of the code for all platforms Berkeley DB supports, and that code lives in the `os` directory of the distribution. Where different code is required, the code is either conditionally compiled or an entirely different version is written. For example, VxWorks versions of some of these files can be found in the distribution directory os_vxworks, and Windows versions can be found in os_windows.

Historically, there are only two difficult questions to answer for each new port. The first question is how to handle shared memory. In order to write multiprocess database applications (not multithreaded, but threads of control running in different address spaces), Berkeley DB must be able to name pieces of shared memory and access them from multiple processes. On UNIX/POSIX systems, we use **mmap** and **shmget** for that purpose, but any interface that provides access to named shared memory is sufficient. If you have a simple, flat address space, you should be able to use the code in `os_vxworks/os_map.c` as a starting point for the port. If you are not intending to write multiprocess database applications, then this won't be necessary, as Berkeley DB can simply allocate memory from the heap if all threads of control will live in a single address space.

The second question is mutex support. Berkeley DB requires some form of **self-blocking** mutual exclusion mutex. Blocking mutexes are preferred as they tend to be less CPU-expensive and less likely to cause thrashing. If blocking mutexes are not available, however, test-and-set will work as well. The code for mutexes is in two places in the system: the include file `dbinc/mutex.h`, and the distribution directory `mutex`.

Berkeley DB uses the GNU autoconf tools for configuration on almost all of the platforms it supports. Specifically, the include file `db_config.h` configures the Berkeley DB build. The simplest way to begin a port is to configure and build Berkeley DB on a UNIX or UNIX-like system, and then take the `Makefile` and `db_config.h` file created by that configuration, and modify it by hand to reflect the needs of the new architecture. Unless you're already familiar with the GNU autoconf toolset, we don't recommend you take the time to integrate your changes back into the Berkeley DB autoconfiguration framework. Instead, send us context diffs of your changes and any new source files you created, and we'll integrate the changes into our source tree.

Finally, we're happy to work with you on the port, or potentially, do the port ourselves, if that is of interest to you. Regardless, if you have any porting questions, just let us know, and we will be happy to answer them.

Source code layout

Directory	Description
LICENSE	Berkeley DB License
btree	Btree access method source code
build_brew	BREW build directory
build_s60	S60 build directory
build_unix	UNIX build directory
build_vxworks	VxWorks build directory.
build_wince	Windows CE build directory.
build_windows	Windows build directory.
clib	C library replacement functions
common	Common Berkeley DB functions
crypto	Cryptographic support
cxx	C++ API
db	Berkeley DB database support
db185	Berkeley DB version 1.85 compatibility API
db_archive	The db_archive utility
db_checkpoint	The db_checkpoint utility
db_deadlock	The db_deadlock utility
db_dump	The db_dump utility
db_dump185	The db_dump185 utility
db_hotbackup	The db_hotbackup utility
db_load	The db_load utility
db_printlog	The db_printlog debugging utility
db_recover	The db_recover utility
db_sql	The db_sql utility
db_stat	The db_stat utility
db_upgrade	The db_upgrade utility
db_verify	The db_verify utility
dbinc	C language include files
dbinc_auto	Automatically generated C language include files
dbm	The dbm/ndbm compatibility APIs
dbreg	Berkeley DB database handle logging support
dist	Berkeley DB administration/distribution tools

Directory	Description
docs	Documentation
docs_src	API and Reference Guide documentation sources
env	Berkeley DB environment support
examples_c	C API example programs
examples_cxx	C++ API example programs
examples_java	Java API example programs
fileops	File object operation support
hash	Hash access method
hmac	Checksum support
hsearch	The hsearch compatibility API
java	Java API
libdb_java	The libdb_java shared library
lock	Lock manager
log	Log manager
mod_db4	Apache module support
mp	Shared memory buffer pool
mutex	Mutexes
os	POSIX 1003.1 operating-system specific functionality
os_brew	BREW operating-system specific functionality
os_qnx	QNX operating-system specific functionality
os_s60	S60 operating-system specific functionality
os_vxworks	VxWorks operating-system specific functionality
os_windows	Windows operating-system specific functionality
perl	DB_File and BerkeleyDB Perl modules
php_db4	PHP module support
qam	Queue access method source code
rep	Replication source code
repmgr	Replication Manager Framework source code
sequence	Sequence source code
tcl	Tcl API
test	Test suite
test_micro	Micro-benchmark test suite

Directory	Description
txn	Transaction manager
xa	X/Open Distributed Transaction Processing XA support

Chapter 48. Additional References

Additional references

For more information on Berkeley DB or on database systems theory in general, we recommend the following sources:

Technical Papers on Berkeley DB

These papers have appeared in refereed conference proceedings, and are subject to copyrights held by the conference organizers and the authors of the papers. Oracle makes them available here as a courtesy with the permission of the copyright holders.

Berkeley DB **(**Postscript **[bdb_usenix.pdf])**

Michael Olson, Keith Bostic, and Margo Seltzer, Proceedings of the 1999 Summer Usenix Technical Conference, Monterey, California, June 1999. This paper describes recent commercial releases of Berkeley DB, its most important features, the history of the software, and Sleepycat Software's Open Source licensing policies.

Challenges in Embedded Database System Administration **(**HTML **[embedded.html])**

Margo Seltzer and Michael Olson, First Workshop on Embedded Systems, Cambridge, Massachusetts, March 1999. This paper describes the challenges that face embedded systems developers, and how Berkeley DB has been designed to address them.

LIBTP: Portable Modular Transactions for UNIX **(**Postscript **[libtp_usenix.pdf])**

Margo Seltzer and Michael Olson, USENIX Conference Proceedings, Winter 1992. This paper describes an early prototype of the transactional system for Berkeley DB.

A New Hashing Package for UNIX **(**Postscript **[hash_usenix.pdf])**

Margo Seltzer and Oz Yigit, USENIX Conference Proceedings, Winter 1991. This paper describes the Extended Linear Hashing techniques used by Berkeley DB.

Background on Berkeley DB Features

These papers, although not specific to Berkeley DB, give a good overview of the way different Berkeley DB features were implemented.

Operating System Support for Database Management

Michael Stonebraker, Communications of the ACM 24(7), 1981, pp. 412-418.

Dynamic Hash Tables

Per-Ake Larson, Communications of the ACM, April 1988.

Linear Hashing: A New Tool for File and Table Addressing

Witold Litwin [witold.html], Proceedings of the 6th International Conference on Very Large Databases (VLDB), 1980

The Ubiquitous B-tree

Douglas Comer, ACM Comput. Surv. 11, 2 (June 1979), pp. 121-138.

Prefix B-trees

Bayer and Unterauer, ACM Transactions on Database Systems, Vol. 2, 1 (March 1977), pp. 11-26.

The Art of Computer Programming Vol. 3: Sorting and Searching

D.E. Knuth, 1968, pp. 471-480.

Document Processing in a Relational Database System

Michael Stonebraker, Heidi Stettner, Joseph Kalash, Antonin Guttman, Nadene Lynn, Memorandum No. UCB/ERL M82/32, May 1982.

Database Systems Theory

These publications are standard reference works on the design and implementation of database systems. Berkeley DB uses many of the ideas they describe.

Transaction Processing Concepts and Techniques

by Jim Gray and Andreas Reuter, Morgan Kaufmann Publishers. We recommend chapters 1, 4 (skip 4.6, 4.7, 4.9, 4.10 and 4.11), 7, 9, 10.3, and 10.4.

An Introduction to Database Systems, Volume 1

by C.J. Date, Addison Wesley Longman Publishers. In the 5th Edition, we recommend chapters 1, 2, 3, 16 and 17.

Concurrency Control and Recovery in Database Systems

by Bernstein, Goodman, Hadzilaco. Currently out of print, but available from http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx.