

Oracle Berkeley DB

***Programmer's Reference
Guide***

12c Release 1

Library Version 12.1.6.0



Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at: <http://www.oracle.com/technetwork/database/berkeleydb/downloads/oslicense-093458.html>

Oracle, Berkeley DB, and Sleepycat are trademarks or registered trademarks of Oracle. All rights to these marks are reserved. No third-party use is permitted without the express prior written consent of Oracle.

Other names may be trademarks of their respective owners.

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at: <https://forums.oracle.com/forums/forum.jspa?forumID=271>

Published 1/24/2014

Table of Contents

Preface	xii
Conventions Used in this Book	xii
For More Information	xii
Contact Us	xiii
1. Introduction	1
An introduction to data management	1
Mapping the terrain: theory and practice	1
Data access and data management	2
Relational databases	3
Object-oriented databases	4
Network databases	4
Clients and servers	5
What is Berkeley DB?	6
Data Access Services	7
Data management services	7
Design	8
What Berkeley DB is not	8
Berkeley DB is not a relational database	9
Berkeley DB is not an object-oriented database	10
Berkeley DB is not a network database	10
Berkeley DB is not a database server	11
Do you need Berkeley DB?	11
What other services does Berkeley DB provide?	12
What does the Berkeley DB distribution include?	12
Where does Berkeley DB run?	12
The Berkeley DB products	13
Berkeley DB Data Store	14
Berkeley DB Concurrent Data Store	14
Berkeley DB Transactional Data Store	14
Berkeley DB High Availability	15
2. Access Method Configuration	16
What are the available access methods?	16
Btree	16
Hash	16
Heap	16
Queue	16
Recno	16
Selecting an access method	16
Btree or Heap?	17
Disk Space Usage	17
Record Access	17
Record Creation/Deletion	18
Cursor Operations	18
Which Access Method Should You Use?	19
Hash or Btree?	19
Queue or Recno?	20

Logical record numbers	21
General access method configuration	23
Selecting a page size	23
Selecting a cache size	24
Selecting a byte order	25
Duplicate data items	26
Non-local memory allocation	27
Btree access method specific configuration	27
Btree comparison	27
Btree prefix comparison	29
Minimum keys per page	30
Retrieving Btree records by logical record number	30
Compression	32
Custom compression	33
Programmer Notes	36
Hash access method specific configuration	37
Page fill factor	37
Specifying a database hash	37
Hash table size	37
Heap access method specific configuration	38
Queue and Recno access method specific configuration	38
Managing record-based databases	38
Record Delimiters	38
Record Length	39
Record Padding Byte Value	39
Selecting a Queue extent size	39
Flat-text backing files	40
Logically renumbering records	40
3. Access Method Operations	42
Database open	42
Opening multiple databases in a single file	43
Configuring databases sharing a file	43
Caching databases sharing a file	43
Locking in databases based on sharing a file	44
Partitioning databases	44
Specifying partition keys	44
Partitioning callback	46
Placing partition files	48
Retrieving records	48
Storing records	49
Deleting records	49
Database statistics	49
Database truncation	50
Database upgrade	50
Database verification and salvage	50
Flushing the database cache	51
Database close	51
Secondary indexes	52
Error Handling With Secondary Indexes	56

Foreign key indexes	56
Cursor operations	59
Retrieving records with a cursor	60
Cursor position flags	60
Retrieving specific key/data pairs	60
Retrieving based on record numbers	61
Special-purpose flags	61
Storing records with a cursor	62
Deleting records with a cursor	64
Duplicating a cursor	64
Equality Join	64
Example	66
Data item count	68
Cursor close	68
4. Access Method Wrapup	69
Data alignment	69
Retrieving and updating records in bulk	69
Bulk retrieval	69
Bulk updates	71
Bulk deletes	71
Partial record storage and retrieval	72
Storing C/C++ structures/objects	74
Retrieved key/data permanence for C/C++	75
Error support	76
Cursor stability	77
Database limits	77
Disk space requirements	78
Btree	78
Hash	79
BLOB support	80
The BLOB threshold	81
Creating BLOBs	81
BLOB access	81
BLOB storage	84
Specifying a Berkeley DB schema using SQL DDL	84
Access method tuning	85
Access method FAQ	86
5. Java API	90
Java configuration	90
Compatibility	91
Java programming notes	91
Java FAQ	92
6. C# API	95
Compatibility	96
7. Standard Template Library API	97
Dbstl introduction	97
Standards compatible	97
Performance overhead	97
Portability	97

Dbstl typical use cases	98
Dbstl examples	98
Berkeley DB configuration	100
Registering database and environment handles	101
Truncate requirements	101
Auto commit support	102
Database and environment identity checks	102
Products, constructors and configurations	102
Using advanced Berkeley DB features with dbstl	103
Using bulk retrieval iterators	103
Using the DB_RMW flag	104
Using secondary index database and secondary containers	104
Using transactions in dbstl	104
Using dbstl in multithreaded applications	105
Working with primitive types	106
Storing strings	107
Store and Retrieve data or objects of complex types	108
Storing varying length objects	108
Storing by marshaling objects	108
Using a DbstlDbt wrapper object	109
Storing arbitrary sequences	110
The SequenceLenFunc function	110
The SequenceCopyFunc function	111
Notes	111
Dbstl persistence	111
Direct database get	111
Change persistence	113
Object life time and persistence	113
Dbstl container specific notes	115
db_vector specific notes	115
Associative container specific notes	116
Using dbstl efficiently	116
Using iterators efficiently	116
Using containers efficiently	117
Dbstl memory management	118
Freeing memory	118
Type specific notes	118
DbEnv/Db	118
DbstlDbt	119
Dbstl miscellaneous notes	119
Special notes about trivial methods	119
Using correct container and iterator public types	120
Dbstl known issues	120
8. Berkeley DB Architecture	122
The big picture	122
Programming model	125
Programmatic APIs	125
C	125
C++	125

STL	126
Java	127
Dbm/Ndbm, Hsearch	127
Scripting languages	127
Perl	127
PHP	127
Tcl	127
Supporting utilities	127
9. The Berkeley DB Environment	130
Database environment introduction	130
Creating a database environment	130
Sizing a database environment	132
Opening databases within the environment	134
Error support	135
DB_CONFIG configuration file	136
File naming	136
Specifying file naming to Berkeley DB	137
Filename resolution in Berkeley DB	137
Examples	138
Shared memory regions	139
Security	140
Encryption	141
Remote filesystems	143
Environment FAQ	143
10. Berkeley DB Concurrent Data Store Applications	145
Concurrent Data Store introduction	145
Handling failure in Data Store and Concurrent Data Store applications	147
Architecting Data Store and Concurrent Data Store applications	148
11. Berkeley DB Transactional Data Store Applications	152
Transactional Data Store introduction	152
Why transactions?	152
Terminology	152
Handling failure in Transactional Data Store applications	153
Architecting Transactional Data Store applications	154
Opening the environment	159
Opening the databases	162
Recoverability and deadlock handling	164
Atomicity	168
Isolation	169
Degrees of isolation	172
Snapshot Isolation	172
Transactional cursors	173
Nested transactions	176
Environment infrastructure	177
Deadlock detection	178
Checkpoints	179
Database and log file archival	181
Log file removal	185
Recovery procedures	185

Hot failover	187
Using Recovery on Journaling Filesystems	189
Recovery and filesystem operations	189
Berkeley DB recoverability	190
Transaction tuning	193
Transaction throughput	195
Transaction FAQ	197
12. Berkeley DB Replication	200
Replication introduction	200
Replication environment IDs	201
Replication environment priorities	201
Building replicated applications	202
Replication Manager methods	203
Base API methods	205
Building the communications infrastructure	206
Connecting to a new site	208
Managing Replication Manager group membership	208
Adding sites to a replication group	209
Removing sites from a replication group	210
Primordial startups	210
Upgrading groups	211
Replication views	212
Managing replication directories and files	213
Replication database directory considerations	213
Managing replication internal files	213
Running Replication Manager in multiple processes	214
One replication process and multiple subordinate processes	214
Persistence of local site network address configuration	215
Programming considerations	215
Handling failure	216
Other miscellaneous rules	216
Running replication using the db_replicate utility	216
One replication process and multiple subordinate processes	216
Common use case	218
Avoiding rollback	219
When to consider an integrated replication application	219
Choosing a Replication Manager acknowledgement policy	220
Elections	221
Synchronizing with a master	223
Delaying client synchronization	223
Client-to-client synchronization	224
Blocked client operations	224
Clients too far out-of-date to synchronize	225
Initializing a new site	225
Bulk transfer	226
Transactional guarantees	226
Master leases	230
Changing group size	233
Read your writes consistency	234

Getting a token	234
Token handling	235
Using a token to check or wait for a transaction	235
Clock skew	235
Using Replication Manager message channels	236
DB_CHANNEL	236
Sending messages over a message channel	237
Message Responses	237
Receiving messages	238
Special considerations for two-site replication groups	238
Network partitions	239
Replication FAQ	241
Ex_rep: a replication example	242
Ex_rep_base: a TCP/IP based communication infrastructure	244
Ex_rep_base: putting it all together	245
Ex_rep_chan: a Replication Manager channel example	246
13. Distributed Transactions	248
Introduction	248
Berkeley DB XA Implementation	248
Building a Global Transaction Manager	248
Communicating with multiple Berkeley DB environments	249
Recovering from GTM failure	249
Managing the Global Transaction ID (GID) name space	249
Maintaining state for each distributed transaction.	250
Recovering from the failure of a single environment	250
Recovering from GTM failure	251
XA Introduction	252
Configuring Berkeley DB with the Tuxedo System	253
Update the Resource Manager File in Tuxedo	253
Build the Transaction Manager Server	253
Update the UBBCONFIG File	253
Restrictions on XA Transactions	254
XA: Frequently Asked Questions	255
14. Application Specific Logging and Recovery	257
Introduction to application specific logging and recovery	257
Defining application-specific log records	258
Automatically generated functions	260
Application configuration	263
15. Programmer Notes	266
Signal handling	266
Error returns to applications	266
Globalization Support	268
Message Format	268
Enable Globalization Support	268
Localization Example	269
Environment variables	270
Multithreaded applications	271
Berkeley DB handles	272
Name spaces	273

C Language Name Space	273
Filesystem Name Space	273
Memory-only or Flash configurations	274
Disk drive caches	276
Copying or moving databases	276
Compatibility with historic UNIX interfaces	277
Run-time configuration	278
Performance Event Monitoring	279
Using the DTrace Provider	280
Using SystemTap	280
Example Scripts	280
Performance Events Reference	281
Programmer notes FAQ	286
16. The Locking Subsystem	287
Introduction to the locking subsystem	287
Configuring locking	288
Configuring locking: sizing the system	288
Standard lock modes	290
Deadlock detection	291
Deadlock detection using timers	292
Deadlock debugging	293
Locking granularity	296
Locking without transactions	297
Locking with transactions: two-phase locking	297
Berkeley DB Concurrent Data Store locking conventions	298
Berkeley DB Transactional Data Store locking conventions	298
Locking and non-Berkeley DB applications	300
17. The Logging Subsystem	302
Introduction to the logging subsystem	302
Configuring logging	302
Log file limits	303
18. The Memory Pool Subsystem	305
Introduction to the memory pool subsystem	305
Configuring the memory pool	306
Warming the memory pool	307
The warm_cache() function	311
19. The Transaction Subsystem	313
Introduction to the transaction subsystem	313
Configuring transactions	314
Transaction limits	314
Transaction IDs	314
Cursors	315
Multiple Threads of Control	315
20. Sequences	316
21. Berkeley DB Extensions: Tcl	317
Loading Berkeley DB with Tcl	317
Installing as a Tcl Package	317
Loading Berkeley DB with Tcl	317
Using Berkeley DB with Tcl	318

Tcl API programming notes	318
Tcl error handling	319
Tcl FAQ	320
22. Berkeley DB Extensions	321
Using Berkeley DB with Apache	321
Using Berkeley DB with Perl	322
Using Berkeley DB with PHP	322
23. Dumping and Reloading Databases	325
The db_dump and db_load utilities	325
Dump output formats	325
Loading text into databases	326
24. Additional References	327
Additional references	327
Technical Papers on Berkeley DB	327
Background on Berkeley DB Features	327
Database Systems Theory	328

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.

This document reflects Berkeley DB 12c Release 1, which provides DB library version 12.1.6.0.

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;
```

Note

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](#)
- [Berkeley DB Getting Started with Replicated Applications for C](#)
- [Berkeley DB C API Reference Guide](#)
- [Berkeley DB C++ API Reference Guide](#)
- [Berkeley DB STL API Reference Guide](#)
- [Berkeley DB TCL API Reference Guide](#)

-
- [Berkeley DB Installation and Build Guide](#)
 - [Berkeley DB Upgrade Guide](#)
 - [Berkeley DB Getting Started with the SQL APIs](#)

To download the latest Berkeley DB documentation along with white papers and other collateral, visit <http://www.oracle.com/technetwork/indexes/documentation/index.html>.

For the latest version of the Oracle Berkeley DB downloads, visit <http://www.oracle.com/technetwork/database/berkeleydb/downloads/index.html>.

Contact Us

You can post your comments and questions at the Oracle Technology (OTN) forum for Oracle Berkeley DB at: <https://forums.oracle.com/forums/forum.jspa?forumID=271>, or for Oracle Berkeley DB High Availability at: <https://forums.oracle.com/forums/forum.jspa?forumID=272>.

For sales or support information, email to: berkeleydb-info_us@oracle.com You can subscribe to a low-volume email announcement list for the Berkeley DB product family by sending email to: bdb-join@oss.oracle.com

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, cellphones, 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 have discussed database systems in general terms. It is 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](#). 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 is 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), which means it is small enough to run in tightly constrained embedded systems, but yet it can take advantage of gigabytes of memory and terabytes of disk if you are using hardware that has those resources.

Each of Berkeley DB's database files can contain up to 256 terabytes of data, assuming the underlying filesystem is capable of supporting files of that size. Note that Berkeley DB applications often use multiple database files. This means that the amount of data your Berkeley DB application can manage is really limited only by the constraints imposed by your operating system, filesystem, and physical hardware.

Berkeley DB also supports high concurrency, allowing thousands of users to operate on the same database files at the same time.

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 does not 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 does not 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, on UNIX systems 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. On other systems which do not support filesystems with read isolation, Berkeley DB provides a tool for safely copying files.

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. By default 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. However, you can configure Berkeley DB to treat large objects in a special way, so that they are accessed in a more efficient manner. Note that these specially-treated large objects are not confined to the four gigabyte limit used for other database objects. See [BLOB support \(page 80\)](#) for more information.

Berkeley DB is not a relational database

While Berkeley DB does provide a set of optional SQL APIs, usually all access to data stored in Berkeley DB is performed using the traditional Berkeley DB APIs.

The traditional Berkeley DB APIs are the way that most Berkeley DB users will use Berkeley DB. Although the interfaces are fairly simple, they are non-standard in that they do not support SQL statements.

That said, Berkeley DB does provide a set of SQL APIs that behave nearly identically to SQLite. By using these APIs you can interface with Berkeley DB using SQL statements. For Unix systems, these APIs are not available by default, while for Windows systems they are available by default. For more information, see the *Berkeley DB Getting Started with the SQL APIs* guide.

Be aware that 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.

Unless Berkeley DB is used with its SQL APIs, it 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 is possible to build a relational system on top of Berkeley DB (indeed, this is what the Berkeley DB SQL API really is). 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.

Berkeley DB is 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.

Berkeley DB is 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.

Berkeley DB is 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 the Berkeley DB Porting Guide 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 the Berkeley DB Installation and Build Guide for more information.

The Berkeley DB distribution includes support for QNX Neutrino. See the Berkeley DB Installation and Build Guide for more information.

The Berkeley DB distribution includes support for VxWorks. See the Berkeley DB Installation and Build Guide 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 the Berkeley DB Installation and Build Guide for more information.

The Berkeley DB products

Oracle provides four Berkeley DB products, each differing by the level of database support that they offer.

- Berkeley DB Data Store
- Berkeley DB Concurrent Data Store
- Berkeley DB Transactional Data Store
- Berkeley DB High Availability

Each product provides additional functionality to the product that precedes it in the list. As a result, you can download Berkeley DB and build an application that provides read-only database access for a single-user, and later add support for more complex database access patterns for multiple users.

The single Open Source distribution of Berkeley DB from Oracle includes the four products and building the distribution automatically builds all four products. However, you must use the same Berkeley DB product throughout an application or group of applications.

To redistribute Berkeley DB software, you must have a license for the Berkeley DB product you use. For further details, refer to the licensing information at: <http://www.oracle.com/technetwork/database/berkeleydb/downloads/index.html>

A comparison of the four Berkeley DB product features is provided in the following table.

	Berkeley DB Data Store	Berkeley DB Concurrent Data Store	Berkeley DB Transactional Data Store	Berkeley DB High Availability
What is this product?	Provides indexed, single-reader/	Adds simple locking with	Adds complete ACID transaction	Adds single-master data

	Berkeley DB Data Store	Berkeley DB Concurrent Data Store	Berkeley DB Transactional Data Store	Berkeley DB High Availability
	single-writer embedded data storage	multiple-reader/ single-writer capabilities	support, as well as recovery	replication across multiple physical machines
Ensures recovery operation	No	No	Yes	Yes
Provides Locking feature	No	Yes	Yes	Yes
Provides concurrent read-write access	No	Yes	Yes	Yes
Provides transactional support	No	No	Yes	Yes
Supports the SQL API	No	No	Yes	No
Provides replication support	No	No	No	Yes

Berkeley DB Data Store

The Berkeley DB Data Store product is an embeddable, high-performance data store. This product supports multiple concurrent threads of control, including multiple processes and multiple threads of control within a process. However, Berkeley DB Data Store does not support locking, and hence does not guarantee 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 updates 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. This product provides built-in concurrency and locking feature. Berkeley DB Concurrent Data Store is intended for applications that need support for concurrent updates to a database that is largely used for reading.

Berkeley DB Transactional Data Store

The Berkeley DB Transactional Data Store product adds support for transactions and database recovery. Berkeley DB Transactional Data Store is intended for applications that require industrial-strength database services, including excellent performance under high-concurrency workloads of read and write operations, the ability to commit or roll back multiple changes to the database at a single instant, and the guarantee that in the event of a catastrophic system or hardware failure, all committed database changes are preserved.

Berkeley DB High Availability

The Berkeley DB High Availability product adds support for data replication. A single master system handles all updates, and distributes these updates to multiple replicas. The number of replicas depends on the application requirements. 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 five access methods: Btree, Hash, Heap, 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 $O(\text{height})$ time, where *height* is the number of levels in the Btree from the root to the leaf pages. The upper bound on the height is $\log_{base_b} N$, where *base_b* is the smallest number of keys on a page, and *N* is the total number of keys stored.

Inserting unordered data into a Btree can result in pages that are only half-full. 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.

Heap

The Heap access method stores records in a heap file. Records are referenced solely by the page and offset at which they are written. Because records are written in a heap file, compaction is not necessary when deleting records, which allows for more efficient use of space than if Btree is in use. The Heap access method is intended for platforms with constrained disk space, especially if those systems are performing a great many record creation and deletions.

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 Heap access methods, between Btree or Hash, or between Queue and Recno, because each of these pairs offer similar functionality.

Btree or Heap?

Most applications use Btree because it performs well for most general-purpose database workloads. But there are circumstances where Heap is the better choice. This section describes the differences between the two access methods so that you can better understand when Heap might be the superior choice for your application.

Before continuing, it is helpful to have a high level understanding of the operating differences between Btree and Heap.

Disk Space Usage

The Heap access method was developed for use in systems with constrained disk space (such as an embedded system). Because of the way it reuses page space, for some workloads it can be much better than Btree on disk space usage because it will not grow the on-disk database file as fast as Btree. Of course, this assumes that your application is characterized by a roughly equal number of record creations and deletions.

Also, Heap can actively control the space used by the database with the use of the DB->set_heapsize() method. When the limit specified by that method is reached, no additional pages will be allocated and existing pages will be aggressively searched for free space. Also records in the heap can be split to fill space on two or more pages.

Record Access

Btree and Heap are fundamentally different because of the way that you access records in them. In Btree, you access a record by using the record's key. This lookup occurs fairly quickly because Btree places records in the database according to a pre-defined sorting order. Your application is responsible for constructing the key, which means that it is relatively easy for your application to know what key is in use by any given record.

Conversely, Heap accesses records based on their offset location within the database. You retrieve a record in a Heap database using the record's Record ID (RID), which is created when the record is added to the database. The RID is created for you; you cannot specify this yourself. Because the RID is created for you, your application does not have control over the key value. For this reason, retrieval operations for a Heap database are usually performed using secondary databases. You can then use this secondary index to retrieve records stored in your Heap database.

Note that an application's data access requirements grow complex, Btree databases also frequently require secondary databases. So at a certain level of complexity you will be using secondary databases regardless of the access method that you choose.

Secondary databases are described in [Secondary indexes \(page 52\)](#).

Record Creation/Deletion

When Btree creates a new record, it places the record on the database page that is appropriate for the sorting order in use by the database. If Btree can not find a page to put the new record on, it locates a page that is in the proper location for the new record, splits it so that the existing records are divided between the two pages, and then adds the new record to the appropriate page.

On deletion, Btree simply removes the deleted record from whatever page it is stored on. This leaves some amount of unused space ("holes") on the page. Only new records that sort to this page can fill that space. However, once a page is completely empty, it can be reused to hold records with a different sort value.

In order to reclaim unused disk space, you must run the `DB->compact()` method, which attempts to fill holes in existing pages by moving records from other pages. If it is successful in moving enough records, it might be left with entire pages that have no data on them. In this event, the unused pages might be removed from the database (depending on the flags that you provide to `DB->compact()`), which causes the database file to be reduced in size.

Both tree searching and page compaction are relatively expensive operations. Heap avoids these operations, and so is able to perform better under some circumstances.

Heap does not care about record order. When a record is created in a Heap database, it is placed on the first page that has space to store the record. No sorting is involved, and so the overhead from record sorting is removed.

On deletion, both Btree and Heap free space within a page when a record is deleted. However, unlike Btree, Heap has no compaction operation, nor does it have to wait for a record with the proper sort order to fill a hole on a page. Instead, Heap simply reuses empty page space whenever any record is added that will fit into the space.

Cursor Operations

When considering Heap, be aware that this access method does not support the full range of cursor operations that Btree does.

- On sequential cursor scans of the database, the retrieval order of the records is not predictable for Heap because the records are not sorted. Btree, of course, sorts its records so the retrieval order is predictable.
- When using a Heap database, you cannot create new records using a cursor. Also, this means that Heap does not support the `DBC->put()` `DB_AFTER` and `DB_BEFORE` flags. You can, however, update existing records using a cursor.
- For concurrent applications, iterating through the records in a Heap database is not recommended due to performance considerations. This is because there is a good chance that there are a lot of empty pages in the database if you have a concurrent application.

For a Heap database, entire regions are locked when a lock is acquired for a database page. If there is then contention for that region, and a new database page needs to be added, then Berkeley DB simply creates a whole new region. The locked region is then padded with empty pages in order to reach the new region.

The result is that if the last used page in a region is 10, and a new region is created at page 100, then there are empty pages from 11 to 99. If you are iterating with a cursor, then all those empty pages must be examined by the cursor before it can reach the data at page 100.

Which Access Method Should You Use?

Ultimately, you can only determine which access method is superior for your application through performance testing using both access methods. To be effective, this performance testing must use a production-equivalent workload.

That said, there are a few times when you absolutely must use Btree:

- If you want to use bulk put and get operations.
- If having your database clustered on sort order is important to you.
- If you want to be able to create records using cursors.
- If you have multiple threads/processes simultaneously creating new records, and you want to be able to efficiently iterate over those records using a cursor.

But beyond those limitations, there are some application characteristics that should cause you to suspect that Heap will work better for your application than Btree. They are:

- Your application will run in an environment with constrained resources and you want to set a hard limit on the size of the database file.
- You want to limit the disk space growth of your database file, and your application performs a roughly equivalent number of record creations and deletions.
- Inserts into a Btree require sorting the new record onto its proper page. This operation can require multiple page reads. A Heap database can simply reuse whatever empty page space it can find in the cache. Insert-intensive applications will typically find that Heap is much more efficient than Btree, especially as the size of the database increases.

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 \(page 40\)](#) 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(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->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->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 \(page 190\)](#) for more information.

Note

The `db_tuner` utility suggests a page size for btree databases that optimizes cache efficiency and storage space requirements. This utility works only when given a pre-populated database. So, it is useful when tuning an existing application and not when first implementing an application.

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 – both of which are 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. Note that `DB_DUPSORT` automatically turns on the `DB_DUP` flag for you, so you do not have to also set that flag; however, it is not an error to also set `DB_DUP` when configuring for sorted duplicate records.

When configuring sorted duplicate records, you can also specify a custom comparison function 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*.

It is an error to attempt to store identical duplicate data items when duplicates are being stored in a sorted order. Any such attempt results in the error message "Duplicate data items are not supported with sorted data" with a DB_KEYEXIST return code.

Note that you can suppress the error message "Duplicate data items are not supported with sorted data" by using the DB_NODUPDATA flag. Use of this flag does not change the database's basic behavior; storing duplicate data items in a database configured for sorted duplicates is still an error and so you will continue to receive the DB_KEYEXIST return code if you try to do that.

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(DB *dbp, const DBT *a, const DBT *b, size_t *locp)
{
    int ai, bi;
```

```

    locp = NULL;
    /*
     * 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(DB *dbp, const DBT *a, const DBT *b, size_t *locp)
{
    int len;
    u_char *p1, *p2;

    locp = NULL;
    /*
     * 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:

```
size_t
compare_prefix(DB *dbp, const DBT *a, const DBT *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(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(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->get(dbcp, &key, &data, DB_SET)) != 0) {
        dbp->err(dbp, ret, "DBC->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->get(dbcp, &key, &data, DB_GET_RECNO)) != 0) {
    dbp->err(dbp, ret, "DBC->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->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.

Note

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

Note

The default compression function is not guaranteed to reduce the size of the on-disk database in every case. It has been tested and shown to work well with English-language text. Of course, in order to determine if the default compression algorithm is beneficial for your application, it is important to test both the final size and the performance using a representative set of data and access patterns.

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 its 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 linearly 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 \(page 69\)](#) 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.

Heap access method specific configuration

Configuring the Heap access method is fairly simple. Beyond the general configuration required for any access method, you can configure how large the database will become, as well as the amount by which the database grows.

If you provide no configuration relative to the heap size, then the database will grow without bound. Whether this is desirable depends on how much disk space is available to your application.

You can limit the size of the on-disk database file by using the `DB->set_heapsize()` method. If the size specified on this method is reached, then further attempts to insert/update records will fail with a `DB_HEAP_FULL` error message.

Heap databases are organized into regions, and each region is a constant size. The size of the region in a heap database is limited by the page size, the first page of the region contains a bitmap describing the available space on the remaining pages in the region. When the database experiences write contention, a region is added to reduce contention. This means heap databases can grow in size very quickly. In order to control the amount by which the database increases, the size of the region is configurable via `DB->set_heap_regionsize()`.

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 \(page 40\)](#).

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 \(page 40\)](#).

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.

You can set the Queue extent size using the `DB->set_q_extentsize()` method. You can see the current extent size using the `DB->get_q_extentsize()` method.

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 \(page 267\)](#).

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 \(page 267\)](#). 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 \(page 267\)](#) 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()`.

For more information on the access method operations supported by the database handle, see the Database and Related Methods section in the *Berkeley DB C API Reference Guide*.

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 five access methods Berkeley DB supports: `DB_BTREE`, `DB_HASH`, `DB_HEAP`, `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 and Heap databases. Queue and Heap 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 \(page 130\)](#)

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).

Note

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:

Note

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.

Note

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 Heap, Queue and Recno access methods. This flag is required if you are creating a new record in a Heap database.

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.

Note

If you are using the Heap access method and you are creating a new record in the database, then the key that you provide to the DB->put() method should be empty. The DB->put() method will return the record's ID (RID) in the key. The RID is automatically created for you when Heap database records are created.

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 the Upgrading Berkeley DB installations chapter of the Berkeley DB Installation and Build Guide 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](#)):

```
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(DB *secondary, const DBT *pkey, const DBT *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;

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 index 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.

Error Handling With Secondary Indexes

An error return during a secondary update in CDS or DS (which requires an abort in TDS) may leave a secondary index inconsistent in CDS or DS. There are a few non-error returns:

- 0
- `DB_BUFFER_SMALL`
- `DB_NOTFOUND`
- `DB_KEYEMPTY`
- `DB_KEYEXIST`

In the case of any other error return during a secondary update in CDS or DS, delete the secondary indices, recreate them and set the `DB_CREATE` flag to the `DB->associate` method. Some examples of error returns that need to be handled this way are:

- `ENOMEM` - indicating there is insufficient memory to return the requested item
- `EINVAL` - indicating that an invalid flag value or parameter is specified

Note that `DB_RUNRECOVERY` and `DB_PAGE_NOTFOUND` are fatal errors which should never occur during normal use of CDS or DS. If those errors are returned by Berkeley DB when running without transactions, check the database integrity with the `DB->verify` method before rebuilding the secondary indices.

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
getcustid(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().

For more information on the operations supported by the cursor handle, see the Database Cursors and Related Methods section in the *Berkeley DB C API Reference Guide*.

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. For Heap databases, this flag always results in the cursor returning the DB_NOTFOUND error.

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

If used on a database configured for sorted duplicates, this returns the smallest record in the database greater than or equal to the supplied key and data items. If used on a database that is *not* configured for sorted duplicates, this flag behaves identically to DB_GET_BOTH.

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(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->get(dbcp, &key, &data, DB_NEXT)) == 0)
    printf("%.10s : %.10s\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->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(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->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->put(dbcp, &key, &data, DB_KEYLAST)) != 0)
        dbp->err(dbp, ret, "DB->cursor");

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

    return (0);
}

```

Note

If you are using the Heap access method and you are creating a new record in the database, then the key that you provide to the DBC->put() method should be empty. The DBC->put() method will return the record's ID (RID) in the key. The RID is automatically created for you when Heap database records are created.

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 \(page 298\)](#).

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->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->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->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->close(join_curs)) != 0 && ret == 0)
    ret = tret;
if (name_curs != NULL &&
    (tret = name_curs->close(name_curs)) != 0 && ret == 0)
    ret = tret;
if (job_curs != NULL &&
    (tret = job_curs->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](#), [MultipleKeyDataEntry](#), and [MultipleRecnoDataEntry](#).

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.

Note that if you want to use a cursor for bulk retrieval of records in a Btree database, you should open the cursor using the DB_CURSOR_BULK flag. This optimizes the cursor for bulk retrieval.

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

```
int
rec_display(DB *dbp)
{
    DBC *dbcp;
    DBT key, data;
    size_t retklen, retklen;
    void *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, DB_CURSOR_BULK))
        != 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->get(dbcp,
            &key, &data, DB_MULTIPLE_KEY | DB_NEXT)) != 0) {
            if (ret != DB_NOTFOUND)
                dbp->err(dbp, ret, "DBcursor->get");
            break;
        }
    }
}
```

```

    }

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

if ((t_ret = dbcp->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 **ABCDEFGHJKLM**, 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.
```

```
ABCDEFGHJIJ0123456789 -> abcdefghijABCDEFGHJIJ0123456789
```

```
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.

```
ABCDEFGHJIJ0123456789 -> 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).

```
ABCDEFGHJIJ0123456789 -> ABCDEFGHJIJ0123456789\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, data_buffer[1024];
...
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 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).

Note

In order to store petabytes of data you can use multiple database files.

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 use the `DB->compact()` method to reclaim disk space. Alternatively, you can 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:

$$\text{useful-bytes-per-page} = (\text{page-size} - \text{page-overhead})$$

$$\text{bytes-of-data} = \text{n-records} * (\text{bytes-per-entry} + \text{page-overhead-for-two-entries})$$

$$\text{n-pages-of-data} = \text{bytes-of-data} / \text{useful-bytes-per-page}$$

$$\text{total-bytes-on-disk} = \text{n-pages-of-data} * \text{page-size}$$

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.

$$161646 = 132000000 / 8166$$

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

$$\text{n-pages-of-data} = \text{npairs} / \text{pairs-per-page}$$

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.

BLOB support

Binary Large Objects (BLOB) support is designed for efficient storage of large objects. An object is considered to be large if it is more than a third of the size of a page. Without BLOB support, large objects must be broken up into smaller pieces, and then reassembled and/or disassembled every time the record is read or updated. Berkeley DB BLOB support avoids this assembly/disassembly process by storing the large object in a special directory set aside for the purpose. The data itself is not kept in the database, nor is it placed into the in-memory cache.

BLOBs can only be stored using the data portion of a key/data pair. They are supported only for Btree, Hash, and Heap databases, and only so long as the database is not configured for checksums, encryption, duplicate records, or duplicate sorted records. In addition, the DBT that you use to access the BLOB data cannot be configured as a partial DBT if you want to access the data using the BLOB's streaming interface (introduced below).

Note that if the environment is transactionally-protected, then all access to the BLOB is also transactionally protected.

BLOBs are not supported for environments configured for replication.

The BLOB threshold

The BLOB threshold is a positive integer, in bytes, which indicates how large an object must be before it is considered a BLOB. By default, the BLOB threshold for any given database is 0, which means that no object will ever be considered a BLOB. This means that the BLOB feature is not used by default for Berkeley DB databases.

In order to use the BLOB feature, you must set the BLOB threshold to a non-zero, positive integer value. You do this for a given database using the `DB->set_blob_threshold()` method. Note that this value must be set before you create the database. At any point after database creation time, this method is ignored.

In addition, if you are using an environment, you can change the default threshold for databases created in that environment to something other than 0 by using the `DB_ENV->set_blob_threshold()` method.

You can retrieve the BLOB threshold set for a database using the `DB->get_blob_threshold()`. You can retrieve the default BLOB threshold set for your environment using the `DB_ENV->get_blob_threshold()`.

Creating BLOBs

There are two ways to create a BLOB. Before you can use either mechanism, you must set the BLOB threshold to a non-zero positive integer value (see the previous section for details). Once the BLOB threshold has been set, you create a BLOB using one of the two following mechanisms:

- Configure the DBT used to access the BLOB data (that is, the DBT used for the data portion of the record) with the `DB_DBT_BLOB` flag. This causes the data to be stored as a BLOB regardless of its size, so long as the database otherwise supports BLOBs.
- Alternatively, creating a data item with a size greater than the BLOB threshold will cause that data item to be automatically stored as a BLOB.

BLOB access

BLOBs may be accessed in the same way as other DBT data, so long as the data itself will fit into memory. More likely, you will find it necessary to use the BLOB streaming API to read and write BLOB data. You open a BLOB stream using the `DBC->db_stream()` method, close it with the `DB_STREAM->close()` method, write to it using the `DB_STREAM->write()` method, and read it using the `DB_STREAM->read()` method.

The following example code fragment can be found in your DB distribution at `.../db/examples/c/ex_blob.c`.

```
...

/* Some necessary variable declarations */
DBC *dbc;      /* Cursor handle */
DB_ENV *dbenv; /* Environment handle */
DB *dbp;       /* Database handle */
DB_STREAM *dbs; /* Stream handle */
DB_TXN *txn;    /* Transaction handle */
DBT data, key;  /* DBT handles */
int ret;
db_off_t size;

...

/* Environment creation skipped for brevity's sake */

...

/* Enable blob files and set the size threshold. */
if ((ret = dbenv->set_blob_threshold(dbenv, 1000, 0)) != 0) {
    dbenv->err(dbenv, ret, "set_blob_threshold");
    goto err;
}

...

/* Database and DBT creation skipped for brevity's sake */

...

/*
 * Access the BLOB using the DB_STREAM API.
 */
if ((ret = dbenv->txn_begin(dbenv, NULL, &txn, 0)) != 0) {
    dbenv->err(dbenv, ret, "txn");
    goto err;
}

if ((ret = dbp->cursor(dbp, txn, &dbc, 0)) != 0) {
    dbenv->err(dbenv, ret, "cursor");
    goto err;
}

/*
 * Set the cursor to a blob. Use DB_DBT_PARTIAL with
 * dlen == 0 to avoid getting any blob data.
 */
```

```

data.flags = DB_DBT_USERMEM | DB_DBT_PARTIAL;
data.dlen = 0;
if ((ret = dbc->get(dbc, &key, &data, DB_FIRST)) != 0) {
    dbenv->err(dbenv, ret, "Not a blob");
    goto err;
}
data.flags = DB_DBT_USERMEM;

/* Create a stream on the blob the cursor points to. */
if ((ret = dbc->db_stream(dbc, &dbs, DB_STREAM_WRITE)) != 0) {
    dbenv->err(dbenv, 0, "Creating stream.");
    goto err;
}

/* Get the size of the blob. */
if ((ret = dbs->size(dbs, &size, 0)) != 0) {
    dbenv->err(dbenv, 0, "Stream size.");
    goto err;
}
/* Read from the blob. */
if ((ret = dbs->read(dbs, &data, 0, (u_int32_t)size, 0)) != 0) {
    dbenv->err(dbenv, 0, "Stream read.");
    goto err;
}
/* Write data to the blob, increasing its size. */
if ((ret = dbs->write(dbs, &data, size/2, 0)) != 0) {
    dbenv->err(dbenv, 0, "Stream write.");
    goto err;
}
/* Close the stream. */
if ((ret = dbs->close(dbs, 0)) != 0) {
    dbenv->err(dbenv, 0, "Stream close.");
    goto err;
}
dbs = NULL;
dbc->close(dbc);
dbc = NULL;
txn->commit(txn, 0);
txn = NULL;
free(data.data);
data.data = NULL;

...

/* Handle clean up skipped. */

```

BLOB storage

BLOBs are not stored in the normal database files on disk in the same way as is other data managed by DB. Instead, they are stored as binary files in a special directory set aside for the purpose.

If you are not using environments, this special BLOB directory is created relative to the current working directory from which your application is running. You can modify this default location using the `DB->set_blob_dir()` method, and retrieve the current BLOB directory using `DB->get_blob_dir()`.

If you are using an environment, then by default the BLOB directory is created within the environment's home directory. You can change this default location using `DB_ENV->set_blob_dir()` and retrieve the current default location using `DB_ENV->get_blob_dir()`. (Note that `DB_ENV->get_blob_dir()` can successfully retrieve the BLOB directory only if `DB_ENV->set_blob_dir()` was previously called.)

Note that because BLOBs are stored outside of the Berkeley DB database files, they are not confined by the four gigabyte limit used for Berkeley DB key and data items. The BLOB size limit is system dependent. It can be the maximum value in bytes of a signed 32 bit integer (if the Berkeley DB-defined type `db_off_t` is four bytes in size), or a signed 64 bit integer (if `db_off_t` is eight bytes in size).

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_codegen` 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_codegen` command, giving the DDL as input. **`db_sql_codegen`** 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_codegen` 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_codegen`** is a one-way process; there is no way to automatically incorporate customizations into newly generated code, if you decide to run **`db_sql_codegen`** again.

To learn more about **`db_sql_codegen`**, please consult the `db_sql_codegen` utility manual page in the Berkeley DB C API Reference 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 \(page 16\)](#) 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 \(page 25\)](#) 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 \(page 43\)](#) 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 \(page 27\)](#)

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* and *Building the Java API* in the *Berkeley DB Installation and Build Guide* 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](#) and [Cursor](#) 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](#) 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](#) and [Cursor](#) both return 0 when a get succeeds, [DB_NOTFOUND](#) (page 267) 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](#) is thrown when a Dbt is too small to hold the corresponding key or data item.

An object of type [DeadlockException](#) is thrown when a deadlock would occur.

An object of type [RunRecoveryException](#), a subclass of [DatabaseException](#), is thrown when there is an error that requires a recovery of the database using db_recover utility.

An object of type [IllegalArgumentException](#) a standard Java Language exception, is thrown when there is an error in method arguments.

An object of type [OutOfMemoryError](#) 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 `cursorlist` argument for [Database.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](#). 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

You can use Berkeley DB in your application through the C# API. To understand the application concepts relating to Berkeley DB, see the first few chapters of this manual. For a general discussion on how to build Berkeley DB applications, see the Berkeley DB Getting Started Guides of C or C++. You can also review the example code of C and C++ from the `examples/c` and `examples/cxx` directories. For a description of all the classes, functions, and enumerations of Berkeley DB C# API, see the [Berkeley DB C# API Reference Guide](#).

A separate Visual Studio solution is provided to build the Berkeley DB C# classes, the examples, and the native support library. See Building the C# API in the Berkeley DB Installation and Build Guide for more information.

The C# API requires .NET framework version 2.0 or above, 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_dotnetVERSION.dll`, containing the API, and two native libraries: `libdb_csharpVERSION.dll` and `libdbVERSION.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 libraries. 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 Framework versions 2.0, 3.0, 3.5, and 4.0.

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. All example methods referred to in this chapter can be found in the `StlAdvancedFeaturesExample` class in the `$DbSrc/examples/stl/StlAdvancedFeatures.cpp` file, and you can build the example in `$DbSrc/build_unix` directory like this: `make exstl_advancedfeatures`, where `DbSrc` is the source directory for Berkeley DB.

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 preferred 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 unmarshalling 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<<" square : "<<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 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 BulkRetrievalItrOpt::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 StdAdvancedFeaturesExample::bulk_retrieval_read() method.

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 the `StdAdvancedFeaturesExample::secondary_containers()` method.

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, but 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.
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 `StlAdvancedFeaturesExample::primitive()` method demonstrates more of these examples.

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 `StlAdvancedFeaturesExample::char_star_string_storage()` and `StlAdvancedFeaturesExample::storing_std_strings()` methods.

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 (*ElemCopyFunc)(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 `ElemSizeFunc` 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 `Dbtclass`, 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 `StdAdvancedFeaturesExample::arbitrary_object_storage` method.

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 (*SequenceLenFunc)(const RGB*);
```

and

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

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

The `SequenceLenFunc` function

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

A `SequenceLenFunc` 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 `SequenceLenFunc` 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 the `StdAdvancedFeaturesExample::arbitray_sequence_storage()` method.

- 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 StlAdvancedFeaturesExample::queue_stack() method.

- 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_func()`.

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 enhance 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 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()`.

In addition, before exiting a thread that uses `dbstl` API, you can call the `dbstl_thread_exit()` function to release any Berkeley DB handles if they are not used by other threads. If you do not call the `dbstl_thread_exit()` function or call this function only in some threads, all open Berkeley DB handles will be closed by the `dbstl_exit()` function. You must call the `dbstl_exit()` function before the process exits, to avoid memory leak and database update loss, if you do not have transactions and persistent log files.

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 `StlAdvancedFeaturesExample::arbitrary_object_storage()` and `StlAdvancedFeaturesExample::char_star_string_storage()` methods.

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_vector::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++ 8 STL library. There are two algorithm functions of Microsoft Visual C++ 10 STL library that do have an issue: `partial_sort()` and `partial_sort_copy()`. These are not compatible because they require the dbstl vector iterator to create a new element when updating the current element. Dbstl vector iterator can copy the new content to the current element, but it cannot create a new one. This requirement is not a part of the C++ STL standard specification, and so dbstl's vector iterator does not support it.

Chapter 8. Berkeley DB Architecture

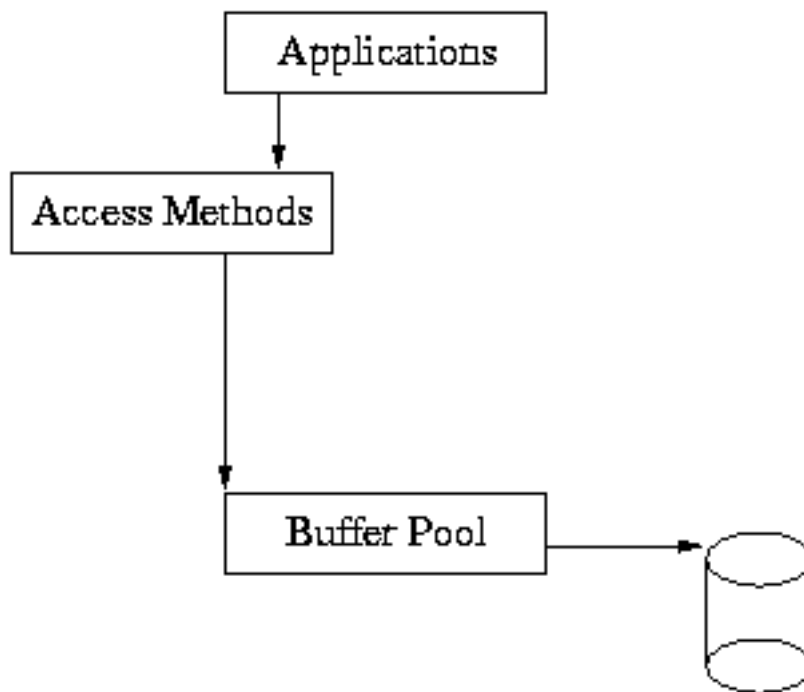
The big picture

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.

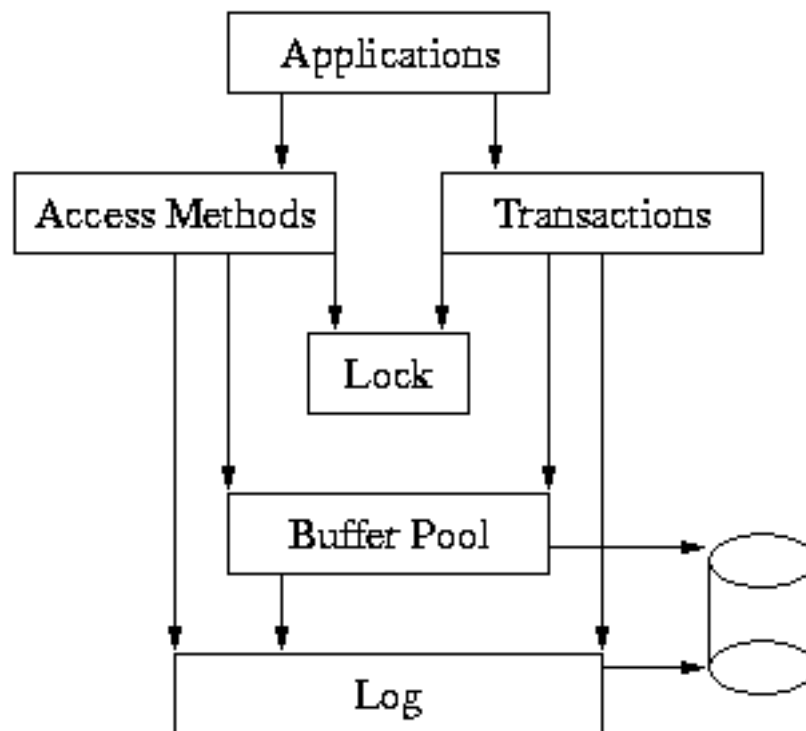
Locking

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 extension --- 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`. See [Using Berkeley DB with Perl \(page 322\)](#) for more information.

PHP

A PHP wrapper is distributed with the Berkeley DB release. See [Using Berkeley DB with PHP \(page 322\)](#) for more information.

Tcl

A Tcl wrapper is distributed with the Berkeley DB release. See [Loading Berkeley DB with Tcl \(page 317\)](#) for more information.

Supporting utilities

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

`db_archive` utility

The `db_archive` utility supports database backup and archival, and log file administration. It facilitates log reclamation and the creation of database snapshots. Generally, some form of log archival must be done if a database environment has been configured for logging or transactions.

db_checkpoint utility

The db_checkpoint utility runs as a daemon process, monitoring the database log and periodically issuing checkpoints. It facilitates log reclamation and the creation of database snapshots. Generally, some form of database checkpointing must be done if a database environment has been configured for transactions.

db_deadlock 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_dump 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_codegen

The db_sql_codegen 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_tuner utility

The db_tuner utility suggests a page size for btree databases that optimizes cache efficiency and storage space requirements.

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.

For more information on the operations supported by the database environment handle, see the Database Environments and Related Methods section in the *Berkeley DB C API Reference Guide*.

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 \(page 145\)](#) 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(char *home, char *data_dir, FILE *errfp, char *progrname)
{
    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->add_data_dir(dbenv, data_dir)) != 0) {
    dbenv->err(dbenv, ret, "add_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);
}

```

Sizing a database environment

The Berkeley DB environment allocates memory to hold shared structures, either in shared regions or in process data space (if the `DB_PRIVATE` flag is specified). There are three distinct memory regions:

- The memory pool (also known as the database page cache),
- the area containing mutexes, and
- the main region which holds all other shared structures.

The shared structures in the main region are used by the lock, transaction, logging, thread and replicatoin subsystems.

Determining the amount of space allocated for each of these shared structures is dependent upon the structure in question. The sizing of the memory pool is discussed in [Configuring the memory pool \(page 306\)](#). The amount of memory needed for mutexes is calculated from the number of mutexes needed by various subsystems and can be adjusted using the `DB_ENV->mutex_set_increment()` method.

For applications using shared memory (that is, they do not specify `DB_PRIVATE`), a maximum memory size for the main region must be specified or left to default. The maximum memory size is specified using the `DB_ENV->set_memory_max()` method.

The amount of memory needed by an application is dependent on the resources that the application uses. For a very rough estimate, add all of the following together:

1. The environment has an overhead of about 80 kilobytes without statistics enabled or 250 kilobytes with statistics enabled.
2. Identify the amount of space you require for your locks:
 - a. Estimate the number of threads of control that will simultaneously access the environment.
 - b. Estimate the number of concurrency locks that, on average, will be required by each thread. For information on sizing concurrency locks, see [Configuring locking: sizing the system \(page 288\)](#).
 - c. Multiply these two numbers, then multiply by 1/2 to arrive at the number of kilobytes required to service your locks.
3. Estimate the number of open database handles you will use at any given time. For each database handle, there is an overhead of about 1/2 kilobyte.
4. Add 1 kilobyte for each active transaction.

Note that these are very rough guidelines. It is best to overestimate the needs of your applications, because if the memory allocation is exhausted the application must be shutdown to increase the allocation.

The estimate for maximum memory need not be exact. In most situations there is little penalty for over estimating. For systems using memory mapped files for the shared environment, this only allocates the address space in the process to hold the maximum memory. The backing file will only be extended as needed. For systems running with `DB_PRIVATE` specified, the maximum memory serves only as a limit and memory is allocated from the process data space as needed. No maximum need be set for private environments.

For locking and thread information, groups of objects are allocated when needed so that there is less contention in the allocator during performance critical operations. Once allocated to a particular use, this memory will only be used for that structure. To avoid runtime contention, or to ensure a minimum number of a particular type of object, the `DB_ENV-`

>set_memory_init() method can be used. This method can set the initial numbers of particular types of structures to allocate at environment creation time.

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)dbp1->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->add_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 four `DB_ENV` methods that affect file naming:

- The `DB_ENV->add_data_dir()` method specifies a directory to search for database files.
- The `DB_ENV->set_log_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.
- The `DB_ENV->set_metadata_dir()` method specifies the directory in which to create persistent metadata files used by the environment.

These methods are intended to permit applications to customize a file locations for an environment. 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, `add_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->add_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->add_data_dir(dbenv, "/b/data2");
dbenv->add_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:

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).

If this flag is specified, then you cannot open more than a single handle for the environment. 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 because the second attempt to open the environment will fail.

2. If the DB_SYSTEM_MEM flag is specified to DB_ENV->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.

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 the Windows Notes section in the Berkeley DB Installation and Build Guide for more information.

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.**

To encrypt a database, you must configure the database for encryption prior to creating it. If you are using a database environment, you must also configure the environment for encryption. In order to create an encrypted database within an environment, you:

1. Configure the environment for encryption using the `DB_ENV->set_encrypt()` method.
2. Open the database environment.
3. Specify the `DB_ENCRYPT` flag to the database handle.
4. Open the database.

Once you have done that, all of the databases that you create in the environment are encrypted/decrypted by the password you specify using the `DB_ENV->set_encrypt()` method.

For databases not created in an environment:

1. Specify the `DB_ENCRYPT` flag to the database handle.
2. Call the `DB->set_encrypt()` method.
3. Open the database.

Note that 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 \(page 139\)](#) 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.

Berkeley DB 11g Release 2 supports encryption using Intel's Performance Primitive (IPP) on Linux. This works only on Intel processors. To use Berkeley DB with IPP encryption, you must have IPP installed along with the cryptography extension. The IPP performance is higher in most cases compared to the current AES implementation. See `--with-cryptography` for more information. See the [Intel Documentation](#) for more information on IPP.

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 \(page 153\)](#) and [Architecting Transactional Data Store applications \(page 154\)](#) 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 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. 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 267) 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 267), 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 147).

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 \(page 147\)](#), 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 \(page 267\)](#), 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 \(page 267\)](#), 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 \(page 267\)](#), 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 267), 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](#) 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 \(page 185\)](#).

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 \(page 267\)](#) 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 \(page 267\)](#), 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 \(page 267\)](#), 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 \(page 185\)](#).

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 267), 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).

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 \(page 267\)](#) 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 recovery 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 recovery. 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 \(page 267\)](#) 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, it may be more difficult to specify unique IDs for each thread of control; third, it may be more difficult to 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_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 \(page 267\)](#). 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 \(page 267\)](#), 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 \(page 267\)](#), 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 \(page 267\)](#), 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;)
    continue;
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 \(page 267\)](#) explicitly or not – attempting new Berkeley DB operations after [DB_RUNRECOVERY \(page 267\)](#) 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 267). 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 \(page 267\)](#) 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",
            "chartreux",
            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->put(dbc, &key, &data, DB_KEYLAST)) {
case 0:
    break;
case DB_LOCK_DEADLOCK:
default:
    va_end(ap);

    /* Retry the operation. */
    if ((t_ret = dbc->close(dbc)) != 0) {
        dbenv->err(
            dbenv, t_ret, "dbc->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->close(dbc)) != 0) {
    dbenv->err(dbenv, ret, "dbc->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 \(page 267\)](#) 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",
        "chartreux",
        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.

Note

Berkeley DB backups (archives) can be recovered using machines of differing byte order. That is, a backup taken on a big-endian machine can be used to restore a database residing on a little-endian machine.

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.

When incremental backups are implemented using this procedure, it is important to know that a database copy taken prior to a bulk loading event (that is, a transaction started with the `DB_TXN_BULK` flag) can no longer be used as the target of an incremental backup. This is true because bulk loading omits logging of some record insertions, so these insertions cannot be rolled forward by recovery. It is recommended that a full backup be scheduled following a bulk loading event.

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->add_data_dir()` for additional information on storing database files in separate directories).

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. Set the `DB_HOTBACKUP_IN_PROGRESS` flag in the environment. This affects the behavior of transactions started with the `DB_TXN_BULK` flag.
2. 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, it is critical that database pages be read atomically as described by [Berkeley DB recoverability \(page 190\)](#).

Note that only UNIX based systems are known to support the atomicity of reads. These systems include: Solaris, Mac OSX, HP/UX and various BSD based systems. Linux and Windows based systems do not support atomic filesystem reads directly. The XFS file system supports atomic reads despite the lack of it in Linux. On systems that do not support atomic file system reads, the `db_hotbackup` utility should be used or a tool can be constructed using the `DB_ENV->backup()` method. Alternatively, you can construct a tool using the `db_copy()` method. You can also perform a hot backup of just a single database in your environment using the `DB_ENV->dbbackup()` method.

3. 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).

4. Reset the `DB_HOTBACKUP_IN_PROGRESS` flag.

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 \(page 185\)](#) for more information).

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.

Additionally, when Replication Manager is running the log file is older than the most out of date active site in the replication group.

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 \(page 181\)](#).

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.

Note

Berkeley DB backups (archives) can be recovered using machines of differing byte order. That is, a backup taken on a big-endian machine can be used to restore a database residing on a little-endian machine.

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->add_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.

The procedure described here is not compatible with the concurrent use of the transactional bulk insert optimization (transactions started with the `DB_TXN_BULK` flag). After the bulk optimization is used, the archive must be created again from scratch starting with step 1.

The `db_hotbackup` utility is the preferred way to automate generating a hot failover system. The first step is to run `db_hotbackup` utility without the `-u` flag. This will create hot backup copy of the databases in your environment. After that point periodically running the `db_hotbackup` utility with the `-u` flag will copy the new log files and run recovery on the backup copy to bring it current with the primary environment.

Note that you can also create your own hot backup solution using the `DB_ENV->backup()` or `DB_ENV->dbbackup()` methods.

To implement your own hot fail over system, the steps below can be followed. However, care should be taken on non-UNIX based systems when copying the database files to be sure that they are either quiescent, or that either the `DB_ENV->backup()` or `db_copy()` routine is used to ensure atomic reads of the database pages.

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->add_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.

Using Recovery on Journaling Filesystems

In some cases, the use of meta-data only journaling file systems can lead to log file corruption. The window of vulnerability is quite small, but if the operating system experiences an unclean shutdown while Berkeley DB is creating a new log file, it is possible that upon file system recovery, the system will be in a state where the log file has been created, but its own meta-data has not.

When a log file is corrupted to this degree, normal recovery can fail and your application may be unable to open your environment. Instead, an error something like this is issued when you attempt to run normal recovery on environment open:

```
Ignoring log file: /var/dblog/log.0000000074: magic number
6c73732f, not 40988
Invalid log file: log.0000000074: Invalid argument
PANIC: Invalid argument
process-private: unable to find environment
txn_checkpoint interface requires an environment configured for
the transaction subsystem
```

In this case, it may be possible to successfully recover the environment by ignoring the log file that was being created – to do this, rename the log file with the highest number to a temporary name:

```
mv DBHOME/log.000000XXX my-temporary-log-file
```

and try running normal environment recovery again. If recovery is successful, and your application is able to open the environment, then you can delete the log file that you renamed.

If recovery is not successful, then you must perform a catastrophic recovery from a previous backup.

This situation has been shown to occur when using ext3 in writeback mode, but other journaling filesystems could exhibit similar behavior.

To be absolutely certain of your application's ability to recover your environment in the event of a system crash, either use non-journaling filesystems, or use a journaling filesystem in a safe (albeit slower) configuration, such as ext3 in ordered mode.

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. On Unix systems, this is not a problem, as these 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. Other operating systems, particularly those based on Linux and Windows, do not provide this guarantee and hot backups may not be performed on these systems by reading data from the file system. The `db_hotbackup` utility should be used on these systems.

An additional 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 \(page 85\)](#), 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 \(page 172\)](#) 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 no longer 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_UNCOMMITTED` 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. This may be particularly beneficial for systems containing several hyperthreaded processor cores. 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-posix-mutexes

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-posix-mutexes` 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 \(page 24\)](#).
- 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_WRITE_NOSYNC flags. 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 are 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 was not done on a quiescent disk) that the number is believable.

An implementation of the previous [example test program](#) 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 \(page 267\)](#), 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 \(page 267\)](#), 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 \(page 267\)](#), 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 \(page 267\)](#) errors. Is this 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.

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.

For more information on the Replication Manager operations, see the Replication and Related Methods section in the *Berkeley DB C API Reference Guide*.

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 log 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->add_data_dir()` (or in the default environment home directory, if not using `DB_ENV->add_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 obtaining a `DB_SITE` handle, and then using it to configure the local site. It can optionally obtain one or more `DB_SITE` handles to configure remote sites. 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 \(page 223\)](#).)

DB_SITE

The `DB_SITE` handle is used to configure a site that belongs to the replication group. You can obtain a `DB_SITE` handle by calling the `DB_ENV->repmgr_site()` method. When you do this, you provide the TCP/IP host name and port that the replication site uses for incoming connections.

Once you have the `DB_SITE` handle, you use the `DB_SITE->set_config()` method to configure the handle. One of the things you can configure about the handle is whether it is the local site (using the `DB_LOCAL_SITE` parameter). You must configure one and only one `DB_SITE` handle to be a local site before you start replication.

You can also optionally configure `DB_SITE` handles for remote sites to help Replication Manager start up more efficiently. 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 \(page 208\)](#).

Once you have configured your `DB_SITE` handles, you start replication using `DB_ENV->repmgr_start()`.

When you are shutting down your application, you must use the `DB_SITE->close()` method to close all your open `DB_SITE` handles before you close your environment handles.

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 \(page 226\)](#). The current implementation requires all sites in the replication group to configure the same 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. Heartbeat messages also help clients request missing master changes in the absence of master activity.

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()`.

Replication Manager applications may configure one or more view sites. A view is a full or partial copy of the replicated data that does not otherwise participate in the replication group. For more information, see [Replication views \(page 212\)](#).

Finally, Replication Manager applications can also make use of the Replication Manager's message channels. This allows the various sites in the replication group to pass messages that are tailored to the application's requirements. For more information, see [Using Replication Manager message channels \(page 236\)](#).

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.

Base API applications may configure one or more view sites. A view is a full or partial copy of the replicated data that does not otherwise participate in the replication group. For more information, see [Replication views \(page 212\)](#).

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 master 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_lsn** 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.

Managing Replication Manager group membership

A replication group is a collection of two or more database environments which are configured to replicate with one another. When operating normally, a replication group consists of a master site and one or more read-only sites.

For Replication Manager applications, the sites comprising the replication group are recorded in an internal group membership database, so even if a group member is not available, it counts towards the group's total site count. This matters for certain replication activities, such as holding elections and acknowledging replication messages that require some number of sites to participate in these activities. Replicated applications will often require all sites, or a majority of sites, to participate before the activity can be completed.

Note

If you are configuring your application to keep replication metadata in-memory by specifying the `DB_REP_CONF_INMEM` flag to the `DB_ENV->rep_set_config()` method,

then the internal group membership database is not stored persistently on disk. This severely limits Replication Manager's ability to automatically manage group membership. For more information, including some work-arounds, see [Managing replication directories and files \(page 213\)](#).

Because Replication Manager tracks group members, there are some administrative activities that you should know about when using Berkeley DB replication.

Adding sites to a replication group

To add a site to a replication group, you merely start up the site such that it knows where at least one site in the group is located. The new site then joins the group. When this happens, the new site is recorded in the group membership database.

Note that when you are starting the very first site in the group for the very first time (called *the primordial start up*), there are no other existing sites to help the new site join the group. In fact, a primordial start up actually creates the group. For this reason, there are some slight differences on how to perform a primordial start up. For a description of this, see [Primordial startups \(page 210\)](#).

When you add a site to a replication group, you use the following general procedure:

- Make sure your replication group is operating well enough that write activity can occur.
- Create and open the environment such that it is configured to use replication.
- Use `DB_ENV->repmgr_site()` to obtain a `DB_SITE` handle. Configure this handle for the local site's host and port information when you create the handle. Then, use `DB_SITE->set_config()` to indicate that this is the local site by setting the `DB_LOCAL_SITE` parameter.
- Use `DB_ENV->repmgr_site()` to obtain a second `DB_SITE` handle. Configure this handle with the host and port information for a site that already belongs to the replication group. Then, use `DB_SITE->set_config()` to indicate this site is a "helper" site by setting the `DB_BOOTSTRAP_HELPER` parameter. By configuring a `DB_SITE` handle in this way, your new site will know how to contact the replication group so that it can join the group.
- Start replication as normal by configuring an acknowledgement policy, setting the site's replication priority, and then calling `DB_ENV->repmgr_start()`.

Note that on subsequent start-ups of your replication code, any helper site information you might provide is ignored because the Replication Manager reads the group membership database in order to obtain this information.

Also, be aware that if the new site cannot be added to the group for some reason (because a master site is not available, or because insufficient replicas are running to acknowledge the new site), the attempt to start the new site via `DB_ENV->repmgr_start()` will fail and return `DB_REP_UNAVAIL`. You can then pause and retry the start up attempt until it completes successfully.

You must use the exact same host string and port number to refer to a given site throughout your application and on each of its sites.

Removing sites from a replication group

Elections and message acknowledgements require knowledge of the total number of sites in the group. If a site is shut down, or is otherwise unable to communicate with the rest of the group, it still counts towards the total number of sites in the group. In most cases, this is the desirable behavior.

However, if you are shutting down a site permanently, then you should remove that site from the group. You might also want to remove a site from the group if you are shutting it down temporarily, but nevertheless for a very long period of time (days or weeks). In either case, you remove a site from the group by:

- Make sure your replication group is operating well enough that write activity can occur.
- On one of the sites in your replication group (this does not have to be the master site), use `DB_ENV->repmgr_site()` to obtain a `DB_SITE` handle. Configure this handle with the host and port information of the site that you want to remove.

Note that this step can occur at any site — including the site that you are removing from the group.

- Call the `DB_SITE->remove()` method. This removes the identified site from the group membership database. If this action is not performed on the master site, the client sends a request to the master to perform the operation and awaits confirmation.

A client removing itself can close its database environment any time after the `DB_SITE->remove()` method returns. A site that has been removed by another site can close its database environment any time after the `DB_EVENT_REP_LOCAL_SITE_REMOVED` event is fired.

Note

Upon completing the above procedure, DO NOT call the `DB_SITE->close()` method. After removing (or even attempting to remove) a site from the group using a `DB_SITE` handle, the handle must never be accessed again.

Primordial startups

If you have never started a site in a replication group before, then the group membership database does not exist. In this situation, you must start an initial site and declare it to be the group creator. This causes the site to become the master, create the group membership database, and create a replication group of size 1. After that, subsequent sites can add themselves to the group as described in [Adding sites to a replication group \(page 209\)](#).

Note

It is never incorrect to declare a site the group creator. This is true even well-after the replication group has been established. This is because group creator information is ignored on any site start-up, except for the primordial start-up; that is, a start-up where the group membership database does not exist.

To declare a site as the group creator:

- Create and open the environment such that it is configured to use replication.
- Use `DB_ENV->repmgr_site()` to obtain a `DB_SITE` handle. Configure this handle for the local site's host and port information when you create the handle. Then, use `DB_SITE->set_config()` to indicate that this is the group creator site by setting the `DB_GROUP_CREATOR` parameter.
- Start replication as normal by configuring acknowledgement policies, setting replication priorities for the site, and then calling `DB_ENV->repmgr_start()`.

Upgrading groups

Prior to the Berkeley DB 11.2.5.2 release, replication group membership was managed differently than in the way it is described in the previous sections. For this reason, when you upgrade from older releases of Berkeley DB to 11.2.5.2 or later, the upgrade procedure is different than when upgrading between other releases.

To perform an upgrade that takes you from the old way of managing group membership to the new way of managing group membership (pre-11.2.5.2 to 11.2.5.2 and later), do the following:

- Update your replication code to use the new `DB_SITE` handle and related methods. Recompile and thoroughly test your code to make sure it is production-ready.
- Do the following one production machine at a time. Make sure to do this at the master site LAST.
 1. Shut down the old replication code.
 2. Install the new replication code.
 3. Configure a `DB_SITE` handle for the local site. Use `DB_SITE->set_config()` to indicate that this is a legacy site by setting the `DB_LEGACY` parameter.
 4. Configure a `DB_SITE` handle for *every other site* in the replication group. Set the `DB_LEGACY` parameter for each of these handles.

Please pay careful attention to this step. To repeat: a `DB_SITE` handle **MUST** be configured for **EVERY** site in the replication group.

5. Start replication. The site is upgraded at this point.

Once you have performed this procedure for each production site, making sure to upgrade the master only after every other site has been upgraded, you are done upgrading your replicated application to use the current group membership mechanism.

On subsequent restarts of your replication code, you do not need to specify the `DB_LEGACY` parameter, nor do you need to identify all of the replication group members. However, it is not an error if you do specify this information on subsequent start ups.

Replication views

A **view** is a replication site that maintains a copy of the replication group's data without incurring other overhead from participating in the replication group. Views are useful for applications that require read scalability, are geographically distributed, have slow network connections to some sites, or require only a subset of replicated databases at some sites.

A client is a **participant** in the replication group because it helps to determine the master and contributes to transaction durability. A view receives its copy of the replicated data without participating in the replication group in these other ways. Specifically, a view cannot ever become master, a view does not vote in elections, and a view does not contribute to transaction durability. A view is similar to an unelectable client because neither can become master. They differ because an unelectable client participates in the replication group by voting and contributing to transaction durability.

A **full view** contains a copy of all replicated databases. You define a full view by calling the `DB_ENV->rep_set_view()` method before opening the environment and supplying a `NULL` value for the **partial_func** parameter. You can then complete any other environment configuration, open the environment, and eventually start replication using the `DB_REP_CLIENT` flags value.

A **partial view** contains a subset of the replicated databases. You define a partial view using the same steps used to define a full view, except that you supply a **partial_func** callback to the `DB_ENV->rep_set_view()` method. The partial function uses application-specific logic to determine the names of the database files to replicate. The partial function should set its **result** output parameter to 0 to reject a database file or to a non-zero value to replicate it. Note that named in-memory databases are always replicated to partial views regardless of partial function logic.

The decision about whether to replicate a particular database file to a partial view is made at the time the database file is first created. It is possible to change the partial function to replicate a different set of database files. However, any database file that was already being replicated will continue to be replicated regardless of the new partial function logic because it already exists at that site.

Recovery on a partial view should always be performed using the `DB_RECOVER` flag to the `DB_ENV->open()` method after calling the `DB_ENV->rep_set_view()` method to supply the partial function. You should not use the `db_recover` utility to recover a partial view because it cannot use a partial function.

Defining a site as a view is a permanent decision; once a site is defined as a view it can never be transformed into a participant. This means that on a view site, the `DB_ENV->rep_set_view()` method must always be called before opening the environment.

An existing participant can be permanently demoted to a view at any time by calling the `DB_ENV->rep_set_view()` method before opening its environment and starting replication. Demoting a participant to a view can fail when starting replication and return `DB_REP_UNAVAIL` if the master is unavailable or there are insufficient participants to acknowledge this operation. You can pause and retry starting replication until it succeeds.

A view cannot become master and does not vote in elections, but elections among the participant sites need to know the number of participants in the replication group. Replication

Manager automatically manages elections and tracks the number of participants. In Base API applications, the number of sites in the replication group set by the `DB_ENV->rep_set_nsites()` method should only count participant sites. Similarly, when a participant site calls the `DB_ENV->rep_select()` method, the `nsites` value should only count participant sites. A view should never call the `DB_ENV->rep_select()` method.

A transaction's durability within a replication group is based on its replication to other sites that can potentially take over as master if the current master fails. A view cannot become master, so it cannot contribute to transaction durability. As a result, Replication Manager views do not send acknowledgements. Base API applications should not count messages returning `DB_REP_ISPERM` from a view as contributing to transaction durability.

Replication Manager views participate in replication group-aware log archiving. A view can also be configured as a client-to-client peer of another site in the replication group. If a partial view does not contain the information requested, Replication Manager will redirect the request to the master.

Managing replication directories and files

Replication database directory considerations

If your application is going to locate databases in any directory other than the environment home directory, you need to consider the directory structure for all sites. There are several recommendations to make in this area.

The use of absolute pathnames is strongly discouraged when replication is in use. Absolute pathnames will not work if there is more than one site on a single machine. Replication with absolute pathnames is unlikely to work across different machines unless great care is taken to make sure the entire path is exactly the same on every machine.

If the master uses a data directory, as specified via `DB_ENV->add_data_dir()` or `DB_ENV->set_create_dir()`, it is recommended that you create the same directory structure on all client sites. When the same directory structure appears on a master and the client, replication creates the client databases in the same directory as the master regardless of the local client directory settings. If a master directory is missing on a client, replication decides where to create the client databases by using the client's local directory settings and the Berkeley DB file naming rules as described in [File naming \(page 136\)](#).

Managing replication internal files

Whether you use the Base API or the Replication Manager, replication creates a set of internal files that are normally stored on-disk in your environment home directory. These files contain metadata which is necessary for replication operations, and so you should never delete these files.

You can cause these files to not be stored on disk, but instead to be held entirely in-memory, by specifying the `DB_REP_CONF_INMEM` flag to the `DB_ENV->rep_set_config()` method. Doing this can improve your application's data throughput by avoiding the disk I/O associated with these metadata files. However, in the event that your application is shut down, the contents of these files are lost. This results in some loss of functionality, including an

increased chance that elections will fail, or that the wrong site will win an election. See the `DB_REP_CONF_INMEM` flag description for more information.

Note that turning on `DB_REP_CONF_INMEM` means that Replication Manager cannot store group membership changes persistently. This is because Replication Manager stores group membership information in an internal database, which is held in memory when `DB_REP_CONF_INMEM` is turned on. For this reason, if your Replication Manager application requires replication metadata to be stored in memory, then you must manually identify all the sites in your replication group using the `DB_LEGACY` site configuration attribute. Be aware that this configuration needs to be made permanent. (Normally, `DB_LEGACY` is used only on a temporary basis for the purpose of upgrading old Replication Manager applications.)

Do the following:

1. Shut down all the sites in your replication group.
2. For every site in your replication group:
 - a. Configure a `DB_SITE` handle for the local site. Use `DB_SITE->set_config()` to indicate that this is a legacy site by setting the `DB_LEGACY` parameter.
 - b. Configure a `DB_SITE` handle for *every other site* in the replication group. Set the `DB_LEGACY` parameter for each of these handles.

Please pay careful attention to this step. To repeat: a `DB_SITE` handle **MUST** be configured for **EVERY** site in the replication group.

3. Restart all the sites in the replication group.

Alternatively, you can store persistent environment metadata files, including those required by replication, in a location other than your environment home directory. This is necessary if your environment home directory is on a device that is unstable, because the persistent metadata files cannot be lost or deleted. You do this using the `DB_ENV->set_metadata_dir()` method.

Note that you must configure the handling of your environment metadata consistently across your entire replication group. That is, if you place your replication metadata in-memory on one site, then it must be placed in-memory on all the sites in the group. Similarly, if you place your replication metadata files in a non-standard directory location on one site, then they must be placed in the exact same directory location on all the sites in your group.

Running Replication Manager in multiple processes

Replication Manager supports shared access to a database environment from multiple processes or environment handles.

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. This is called the *replication process*. That is, the replication process is the Replication Manager process that is responsible for initiating and processing most replication messages.

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, and so this becomes the replication process. Replication Manager ignores this step in any subsequent processes, called *subordinate processes*.

If the replication process quits and there are one or more subordinate processes available, one subordinate process will automatically take over as the replication process. During an automatic takeover on the master site, the rest of the replication group briefly delays elections to prevent an unnecessary change of master. Automatic takeover does not occur if the replication process fails unexpectedly. Automatic takeover is most reliable when the `DB_REP_ACK_TIMEOUT` is the same value on all sites in the replication group.

Persistence of local site network address configuration

The local site network address is stored in shared memory, and remains intact even when (all) processes close their environment handles gracefully and terminate. A process which opens an environment handle without running recovery automatically inherits the existing local site network address configuration. Such a process may not change the local site address (although it is allowed to redundantly specify a local site configuration matching that which is already in effect).

In order to change the local site network address, the application must run recovery. The application can then specify a new local site address before restarting Replication Manager. The application should also remove the old local site address from the replication group if it is no longer needed.

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 there are only two replication notification that can occur in a subordinate process:

1. DB_EVENT_REP_PERM_FAILED.
2. DB_EVENT_REP_AUTOTAKEOVER_FAILED

It is not supported for a process running Replication Manager to spawn a subprocess.

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 \(page 153\)](#). To summarize, there are two ways to handle failure of a process:

- 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.
- 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. A subordinate process cannot automatically take over as the replication process if the previous replication process failed. 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.

Running replication using the db_replicate utility

Replication Manager supports shared access to a database environment from multiple processes. Berkeley DB provides a replication-aware utility, db_replicate, that enables you to upgrade an existing Transactional Data Store application, as discussed in the [Transactional Data Store introduction \(page 152\)](#) section, to a replication application with minor modifications. While the db_replicate utility simplifies the use of replication with a Transactional Data Store application, you must still understand replication and its impact on the application.

One replication process and multiple subordinate processes

Based on the terminology introduced in the [Running Replication Manager in multiple processes \(page 214\)](#) section, application processes are subordinate processes and the db_replicate utility is the replication process.

You must consider the following items when planning to use the db_replicate utility in combination with a Transactional Data Store application.

- **Memory regions**
The `db_replicate` utility requires shared memory access among separate processes, and therefore cannot be used with `DB_PRIVATE`.
- **Multi-process implications**
You must understand and accept all of the Transactional Data Store implications of multi-process use as specified in [Architecting Transactional Data Store applications \(page 154\)](#). Special attention should be paid to the coordination needed for unrelated processes to start up correctly.
- **Replication configuration**
Several configuration settings are required for replication. You must set the `DB_INIT_REP` and `DB_THREAD` flags for the `DB_ENV->open()` method. Another required configuration item is the local address. You identify this by creating a `DB_SITE` handle and then setting the `DB_LOCAL_SITE` parameter using the `DB_SITE->set_config()` method. You also tell sites how to contact other sites by creating `DB_SITE` handles for those sites. Most replication configuration options start with reasonable defaults, but applications have to customize at least some of them. You can set all replication related configuration options either programmatically or in the `DB_CONFIG` file.
- **Starting the application and replication**
The `db_replicate` utility assumes that an environment exists and that the application has run recovery, if necessary, and created and configured the environment. The startup flow of a typical Transactional Data Store application may not be the best flow for a replication application and you must understand the issues involved. For instance, if an application starts, runs recovery, and performs update operations before starting the `db_replicate` utility, then if that site becomes a client when replication starts, those update operations will be rolled back.
- **Handling events**
Almost all of the replication-specific events are handled by the `db_replicate` utility process, and therefore the application process does not see them. If the application needs to know the information from those replication-specific events, such as role changes, the application must call the `rep_stat()` method. The one replication-specific event the application can now receive is the `DB_EVENT_REP_PERM_FAILED` event. See [Choosing a Replication Manager acknowledgement policy \(page 220\)](#) for additional information about this event.
- **Handling errors**
There are some error return values that relate only to replication. Specifically, the `DB_REP_HANDLE_DEAD` error should now be handled by the application. Also, if master leases are in use, then the application also needs to consider the `DB_REP_LEASE_EXPIRED` error.
- **Flexibility tradeoff**
You are giving up flexibility for the ease of use of the utility. Application complexity or requirements may eventually dictate integrating replication calls into the application instead of using the `db_replicate` utility.
- **Read-only client application**
The application requires additional changes to manage the read-only status when the application takes on the role of a client.

Common use case

This section lists the steps needed to get replication running for a common use case of the `db_replicate` utility. The use case presented is an existing Transactional Data Store application that already has its environment and databases created and is up and running. At some point, replication is considered because failover protection or balancing the read load may now be desired.

1. To understand the issues involved in a replication application, see the `db_replicate` utility section in the *API Reference Guide*, the Replication Chapter in the *Programmer's Reference Guide*, and the source code of the `ex_rep_mgr` example program.
2. Make a local hot backup of the current application environment to a new location to use as a testing area.
3. Add the `DB_INIT_REP` and `DB_THREAD` flags (if not already being used) to the application or the `DB_CONFIG` file.
4. Modify the `DB_CONFIG` file to add the necessary replication configuration values. At a minimum, the local host and port information must be added using the `repmgr_site` method parameter. As more sites are added to the group, remote host and port information can optionally also be added by adding more `repmgr_site` method parameters to the `DB_CONFIG` file.
5. Rebuild the application and restart it in the current testing directory.
6. Start the `db_replicate` utility on the master site with the `-M` option and any other options needed such as `-h` for the home directory. At this point you have a lone master site running in an environment with no other replicated sites in the group.
7. Optionally, prepare to start a client site by performing a manual hot backup of the running master environment to initialize a client target directory. While replication can make its own copy, the hot backup will expedite the synchronization process. Also, if the application assumes the existence of a database and the client site is started without data, the application may have errors or incorrectly attempt to create the database.
8. Copy the application to the client target.
9. Modify the client environment's `DB_CONFIG` file to set the client's local host and port values and to add remote site information for the master site and any other replication configuration choices necessary.
10. Start the application on the client. The client application should not update data at this point, as explained previously.
11. Start the `db_replicate` utility specifying the client environment's home directory using the `-h` option. Omit the `-M` option in this case, because the utility defaults to starting in the client role.

Once the initial replication group is established, do not use the `-M` option with the `db_replicate` utility. After the initial start, the `db_replicate` utility assumes the use of

elections. If a site crashes, it should rejoin the group as a client so that it can synchronize with the rest of the group.

Avoiding rollback

Depending on how an application is structured, transactional rollback can occur. If this is possible, then you must make application changes or be prepared for successful transactions to disappear. Consider a common program flow where the application first creates and opens the environment with recovery. Then, immediately after that, the application opens up the databases it expects to use. Often an application will use the DB_CREATE flag so that if the database does not exist it is created, otherwise the existing one is used automatically. Then the application begins servicing transactions to write and read data.

When replication is introduced, particularly via the `db_replicate` utility, the possibility of rollback exists unless the application takes steps to prevent it. In the situation described above, if all of the above steps occur before the `db_replicate` utility process starts, and the site is started as a client, then all the operations will be rolled back when the site finds the master. The client site will synchronize with the log and operations on the master site, so any operations that occurred in the client application before it knew it was a client will be discarded.

One way to reduce the possibility of rollback is to modify the application so that it only performs update operations (including creation of a database) if it is the master site. If the application refrains from updating until it is the master, then it will not perform operations when it is in the undefined state before replication has been started. The event indicating a site is master will be delivered to the `db_replicate` utility process, so the application process must look for that information via the `rep_stat()` method. A site that is expecting to perform updates may need to poll via the `rep_stat()` method to see the state change from an undefined role to either the master or client role. Similarly, since a client site cannot create a database, it may need to poll for the database's existence while the client synchronizes with the master until the database is created at the client site.

When to consider an integrated replication application

The `db_replicate` utility provides the means to achieve a replicated application quickly. However, the trade-off for this rapid implementation is that the full flexibility of replication is not available. Some applications may eventually need to consider integrating directly with replication rather than using the `db_replicate` utility if greater flexibility is desired.

One likely reason for considering integration would be the convenience of receiving all replication-related events in the application process and gaining direct knowledge of such things as role changes. Using the event callback is cleaner and easier than polling for state changes via the `rep_stat()` method.

A second likely reason for integrating replication directly into the application is the multi-process aspect of the utility program. The developer may find it easier to insert the start of replication directly into the code once the environment is created, recovered, or opened, and avoid the scenario where the application is running in the undefined state. Also it may simply be easier to start the application once than to coordinate different processes and their startup order in the system.

Choosing a Replication Manager acknowledgement policy

Replication Manager allows the user to choose from a variety of acknowledgement policies. There are two characteristics that should be considered when choosing the policy: consistency and durability. Consistency means making sure some number of clients have applied all available master transactions. Durability, in this context, means only indicating success only if enough clients have applied a transaction. The issue of how many is enough depends on the application's requirements and varies per acknowledgement policy. For example, `DB_REPMGR_ACKS_QUORUM` means the data will survive a change in master or a network partition. In most cases, the number of sites for consistency is equal to the number of sites for durability. Replication Manager uses the consistency value to decide whether or not to wait for acknowledgements. Replication manager uses the durability value to decide either the transaction was successfully processed or that a `DB_EVENT_REP_PERM_FAILED` event should be generated.

Replication Manager also strives to give the application the answer and return to the application as quickly as possible. Therefore, if it knows that the number of sites connected is insufficient to meet the consistency value, then it does not wait for any acknowledgements and if it knows that the durability value cannot be met, it returns `DB_EVENT_REP_PERM_FAILED` immediately to the user.

With one exception, discussed below, all acknowledgement policies combine the consistency and durability values. For most policies the primary purpose is the durability of the data. For example, the `DB_REPMGR_ACKS_QUORUM` policy ensures that, if successful, the transaction's data is safe in the event of a network partition so that a majority of the sites in the group have the data. The `DB_REPMGR_ACKS_NONE` policy does not consider either consistency or durability, and it is very fast because it does not wait for any acknowledgements and it does not ever trigger the `DB_EVENT_REP_PERM_FAILED` event. Other policies, `DB_REPMGR_ACKS_ALL` and `DB_REPMGR_ACKS_ALL_PEERS`, have a primary purpose of consistency. These two policies wait for acknowledgements from all (or all electable) sites in the group.

In the face of failure, however, the `DB_REPMGR_ACKS_ALL` and `DB_REPMGR_ACKS_ALL_PEERS` policies can result in a surprising lack of consistency due to the fact that Replication Manager strives to give the answer back to the application as fast as it can. So, for example, with `DB_REPMGR_ACKS_ALL`, and one site down, Replication Manager knows that disconnected site can never acknowledge, so it immediately triggers `DB_EVENT_REP_PERM_FAILED`. An unfortunate side effect of this policy is that existing, running sites may fall further and further behind the master if the master site is sending a fast, busy stream of transactions and never waiting for any site to send an acknowledgement. The master does not wait because the consistency value cannot be met, and it does trigger the `DB_EVENT_REP_PERM_FAILED` event because the durability value cannot be met, but those actions now affect the consistency of the other running sites.

In order to counteract this unfortunate side effect, the `DB_REPMGR_ACKS_ALL_AVAILABLE` acknowledgement policy focuses on the consistency aspect, but also considers durability. This policy uses all sites for consistency, and a quorum of sites for its decision about durability. As long as there is a non-zero number of client replicas to send to, the master will wait for all available sites to acknowledge the transaction. As long as any client site is connected, this policy will prevent the master from racing ahead if one or more sites is down. On the master,

this policy will then consider the transaction durable if the number of acknowledgements meets quorum for the group.

The following acknowledgement policies determine durability using acknowledgements from electable peers only: `DB_REPMGR_ACKS_QUORUM`, `DB_REPMGR_ACKS_ONE_PEER`, `DB_REPMGR_ACKS_ALL_PEERS`. An electable peer is a site where the priority value is greater than zero. In replication groups using these policies, an unelectable site does not send acknowledgements and cannot contribute to transaction durability.

Elections

Replication Manager automatically conducts elections when necessary, based on configuration information supplied to the `DB_ENV->rep_set_priority()` method, unless the application turns off automatic elections using the `DB_ENV->rep_set_config()` 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 `DB_ENV->rep_elect()` method's `nvotes` argument 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 `DB_REPMGR_CONF_2SITE_STRICT` flag. 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 or more remote sites (called its "peers") to receive client-to-client requests. You do this by setting the `DB_REPMGR_PEER` parameter using the `DB_SITE->set_config()` method. Replication Manager always tries to send requests marked with the `DB_REP_ANYWHERE` flag to a peer, if available. However, a `DB_REP_REREQUEST` message is always handled by requesting the information from the master. A view can serve as a peer, but a partial view is more likely to be missing requested information, which will then be requested from the master.

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 to turn off the `DB_REP_CONF_AUTOINIT` flag. Turning off 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 to turn off the `DB_REP_CONF_AUTOINIT` flag. Turning off 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 \(page 181\)](#). 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 \(page 185\)](#).
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.

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.

The rest of this section discusses transactional guarantee considerations in Base API applications.

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 a Base API 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.

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, or 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 a failure of half of the replication group 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 the failure of at least half of the replication group. If clients grant a lease while holding updated data in cache, and failure occurs, then the data is no longer present on the clients and rollback can occur if a sufficient number of other sites also crash.

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. Base API 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 skew ratio 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).
9. Unelectable (zero priority) sites never grant leases and cannot be used to guarantee data durability. A majority of sites in the replication group must be electable in order to meet the requirement of getting lease grants from a majority of sites. Minimizing the number of unelectable sites improves replication group availability.

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.

Applications using master leases should be prepared to handle DB_REP_LEASE_EXPIRED errors from read operations on a master and from the DB_TXN->commit() method.

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.

Master lease checks cannot succeed until a majority of sites have completed client synchronization. Read operations on a master performed before this condition is met can use the DB_IGNORE_LEASE flag to avoid errors.

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.

Changing group size

If you are using master leases and you change the size of your replication group, there is a remote possibility that you can lose some data previously thought to be durable. This is only true for users of the Base API.

The problem can arise if you are removing sites from your replication group. (You might be increasing the size of your group overall, but if you remove all of the wrong sites you can lose data.)

Suppose you have a replication group with five sites; A, B, C, D and E; and you are using a quorum acknowledgement policy. Then:

1. Master A replicates a transaction to replicas B and C. Those sites acknowledge the write activity.
2. Sites D and E do not receive the transaction. However, B and C have acknowledged the transaction, which means the acknowledgement policy is met and so the transaction is considered durable.
3. You shut down sites B and C. Now only A has the transaction.
4. You decrease the size of your replication group to 3 using `DB_ENV->rep_set_nsites()`.
5. You shut down or otherwise lose site A.
6. Sites D and E hold an election. Because the size of the replication group is 3, they have enough sites to successfully hold an election. However, neither site has the transaction in question. In this way, the transaction can become lost.

An alternative scenario exists where you do not change the size of your replication group, or you actually increase the size of your replication group, but in the process you happen to remove the exact wrong sites:

1. Master A replicates a transaction to replicas B and C. Those sites acknowledge the write activity.
2. Sites D and E do not receive the transaction. However, B and C have acknowledged the transaction, which means the acknowledgement policy is met and so the transaction is considered durable.
3. You shut down sites B and C. Now only A has the transaction.
4. You add three new sites to your replication group: F, G and H, increasing the size of your replication group to 6 using `DB_ENV->rep_set_nsites()`.
5. You shut down or otherwise lose site A before F, G and H can be fully populated with data.
6. Sites D, E, F, G and H hold an election. Because the size of the replication group is 6, they have enough sites to successfully hold an election. However, none of these sites has the transaction in question. In this way, the transaction can become lost.

This scenario represents a race condition that would be highly unlikely to be seen outside of a lab environment. To minimize the chance of this race condition occurring to the absolute minimum, do one or more of the following when using master leases with the Base API:

1. Require all sites to acknowledge transaction commits.
2. Never change the size of your replication group unless all sites in the group are running and communicating normally with one another.
3. Don't remove (or replace) a large percentage of your sites from your replication group unless all sites in the group are running and communicating normally with one another. If you are going to remove a large percentage of your sites from your replication group, try removing just one site at a time, pausing in between each removal to give the replication group a chance to fully distribute all writes before removing the next site.

Read your writes consistency

Some applications require the ability to read replicated data at a client site, and determine whether it is consistent with data that has been written previously at the master site.

For example, a web application may be backed by multiple database environments, linked to form a replication group, in order to share the workload. Web requests that update data must be served by the replication master, but any site in the group may serve a read-only request. Consider a work flow of a series of web requests from one specific user at a web browser: the first request generates a database update, but the second request merely reads data. If the read-only request is served by a replication client database environment, it may be important to make sure that the updated data has been replicated to the client before performing the read (or to wait until it has been replicated) in order to show this user a consistent view of the data.

Berkeley DB supports this requirement through the use of transaction "tokens". A token is a form of identification for a transaction within the scope of the replication group. The application may request a copy of the transaction's token at the master site during the execution of the transaction. Later, the application running on a client site can use a copy of the token to determine whether the transaction has been applied at that site.

It is the application's responsibility to keep track of the token during the interim. In the web example, the token might be sent to the browser as a "cookie", or stored on the application server in the user's session context.

The operations described here are supported both for Replication Manager applications and for applications that use the replication Base API.

Getting a token

In order to get a token, the application must supply a small memory buffer, using the `DB_TXN->set_commit_token()` method.

Note that a token is generated only upon a successful commit operation, and therefore the token buffer content is valid only after a successful commit. Also, if a transaction does not perform any update operations it does not generate a useful token.

In the Berkeley DB Java and C# API, getting a token is simpler. The application need only invoke the [Transaction.getCommitToken\(\)](#) method, after the transaction has committed.

Token handling

The application should not try to interpret the content of the token buffer, but may store and/or transmit it freely between systems. However, since the buffer contains binary data it may be necessary to apply some encoding for transmission (e.g., base 64).

The data is resilient to differences in byte order between different systems. It does not expire: it may be retained indefinitely for later use, even across Berkeley DB version upgrades.

Using a token to check or wait for a transaction

The `DB_ENV->txn_applied()` method takes a copy of a token, and determines whether the corresponding transaction is currently applied at the local site. The timeout argument allows the application to block for a bounded amount of time for cases where the transaction has not yet been applied.

Depending on the transaction durability levels implemented or configured by the application, it is sometimes possible for a transaction to disappear from a replication group if an original master site fails and a different site becomes the new master without having received the transaction. When the `DB_ENV->txn_applied()` method discovers this, it produces the `DB_NOTFOUND` return code.

This means that the results of `DB_ENV->txn_applied()` are not guaranteed forever. Even after a successful call to `DB_ENV->txn_applied()`, it is possible that by the time the application tries to read the data, the transaction and its data could have disappeared.

To avoid this problem the application should do the read operations in the context of a transaction, and hold the transaction handle open during the `DB_ENV->txn_applied()` call. The `DB_ENV->txn_applied()` method itself does not actually execute in the context of the transaction; but no rollbacks due to new master synchronization ever occur while a transaction is active, even a read-only transaction at a client site.

Note that the `DB_ENV->txn_applied()` method can return `DB_LOCK_DEADLOCK`. The application should respond to this situation just as it does for any other normal operation: abort any existing transaction, and then pause briefly before retrying.

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.

Using Replication Manager message channels

The various sites comprising a replication group frequently need to communicate with one another. Mostly, these messages are handled for you internally by the Replication Manager. However, your application may have a requirement to pass messages beyond what the Replication Manager requires in order to satisfy its own internal workings.

For this reason, you can access and use the Replication Manager's internal message channels. You do this by using the `DB_CHANNEL` class, and by implementing a message handling function on each of your sites.

Note that an example of using Replication Manager message channels is available in the distribution. See [Ex_rep_chan: a Replication Manager channel example \(page 246\)](#) for more information.

DB_CHANNEL

The `DB_CHANNEL` class provides a series of methods which allow you to send messages to the other sites in your replication group. You create a `DB_CHANNEL` handle using the `DB_ENV->repmgr_channel()` method. When you are done with the handle, close it using the `DB_CHANNEL->close()` method. A closed handle must never be accessed again. Note that all

channel handles should be closed before the associated environment handle is closed. Also, allow all message operations to complete on the channel before closing the handle.

When you create a DB_CHANNEL handle, you indicate what channel you want to use. Possibilities are:

- The numerical env ID of a remote site in the replication group.
- DB_EID_MASTER

Messages sent on this channel are sent only to the master site. Note that messages are always sent to the current master, even if the master has changed since the channel was opened.

If the local site is the master, then sending messages on this channel will result in the local site receiving those messages echoed back to itself.

Sending messages over a message channel

You can send any message you want over a message channel. The message can be as simple as a character string and as complex as a large data structure. However, before you can send the message, you must encapsulate it within one or more DBTs. This means [marshaling the message](#) if it is contained within a complex data structure.

The methods that you use to send messages all accept an array of DBTs. This means that in most circumstances it is perfectly acceptable to send multi-part messages.

Messages may be sent either asynchronously or synchronously. To send a message asynchronously, use the DB_CHANNEL->send_msg() method. This method sends its message and then immediately returns without waiting for any sort of a response.

To send a message synchronously, use the DB_CHANNEL->send_request() method. This method blocks until it receives a response from the site to which it sent the message (or until a timeout threshold is reached).

Message Responses

Message responses are required if a message is sent on a channel using the DB_CHANNEL->send_request() method. That method accepts the address of a single DBT which is used to receive the response from the remote site.

Message responses are encapsulated in a single DBT. The response can be anything from a complex data structure, to a string, to a simple type, to no information at all. In the latter case, receipt of the DBT is sufficient to indicate that the request was received at the remote site.

Responses are sent back from the remote system using its message handling function. Usually that function calls DB_CHANNEL->send_msg() to send a single response.

The response must be contained in a single DBT. If a multi-part response is required by the application, you can configure the response DBT that you provide to DB_CHANNEL->send_request() for [bulk operations](#).

Receiving messages

Messages received at a remote site are handled using a callback function. This function is configured for the local environment using the `DB_ENV->repmgr_msg_dispatch()` method. For best results, the message dispatch function should be configured for the local environment before replication is started. In this way, you do not run the risk of missing messages sent after replication has started but before the message dispatch function is configured for the environment.

The callback configured by `DB_ENV->repmgr_msg_dispatch()` accepts four parameters of note:

- A response channel. This is the channel the function will use to respond to the message, if a response is required. To respond to the message, the function uses the `DB_CHANNEL->send_msg()` method.
- An array of DBTs. These hold the message that this function must handle.
- A numerical value that indicates how many elements the previously described array holds.
- A flag that indicates whether the message requires a response. If the flag is set to `DB_REPMGR_NEED_RESPONSE`, then the function should send a single DBT in response using the channel provided to this function, and the `DB_CHANNEL->send_msg()` method.

For an example of using this callback, see the `operation_dispatch()` function, which is available with the [ex_rep_chan example](#) in your product distribution.

Special considerations for two-site replication groups

One of the benefits of replication is that it helps your application remain available for writes even when a site crashes. Another benefit is the added durability achieved by storing multiple copies of your application data at different sites. However, if your replication group contains only two sites, you must prioritize which of these benefits is more important to your application.

A two-site replication group is particularly vulnerable to duplicate masters if there is a loss of communication between the sites. The original master continues to accept new transactions. If the original client detects the loss of the master and elects itself master, it also starts accepting new transactions. When communications are restored, there are duplicate masters and one site's new transactions will be rolled back.

If it is unacceptable to your application for any new transactions to be rolled back, the alternative in a two-site replication group is to require both sites to be present in order to elect a master. This stops a client from electing itself master when it loses contact with the master and prevents creation of parallel sets of transactions, one of which must be rolled back.

However, requiring both sites to be present to elect a master results in a loss of write availability when the master crashes. The client cannot take over as master and the replication group exists in a read-only state until the original master site rejoins the replication group.

Replication Manager applications use the `DB_ENV->rep_set_config()` method `DB_REPMGR_CONF_2SITE_STRICT` flag to make this tradeoff between write availability and transaction durability. When this flag is turned on, Replication Manager favors transaction durability. When it is turned off, Replication Manager favors write availability.

A two-site Replication Manager application that uses heartbeats in an environment with frequent communications disruptions generally should operate with the `DB_REPMGR_CONF_2SITE_STRICT` flag turned on. Otherwise, frequent heartbeat failures will cause frequent duplicate masters and the resulting elections and client synchronizations will make one or both sites unavailable for extended periods of time.

Base API applications use the values of the `nvotes` and `nsites` parameters in calls to the `DB_ENV->rep_elect()` method to make this tradeoff. For more information, see [Elections \(page 221\)](#).

A replication group containing only two electable sites is subject to duplicate masters and rollback of one site's new transactions even when it contains additional unelectable sites. The `DB_REPMGR_CONF_2SITE_STRICT` does not apply in this case because the replication group is larger than two sites.

If both write availability and transaction durability are important to your application, you should strongly consider having three or more electable sites in your replication group. You should also carefully choose an acknowledgement policy that requires at least a quorum of sites. It is best to have an odd number of electable sites to provide a clear majority in the event of a network partition.

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`. Replication Manager applications automatically handle duplicate master situations. If a Base API 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 automatically maintains the number of sites in the replication group, so this technique is only available to Base API applications.

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 observing a strict majority rule that prevents the survivor from taking over. Thus it avoids multiple masters and the need to unroll some transactions if both sites are running but cannot communicate. But it does leave the group in a read-only state until both sites are available. If application availability while one site is down is a priority and it is acceptable to risk unrolling some transactions, there is a configuration option to turn off the strict majority rule and allow the surviving client to declare itself to be master. 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. A Replication Manager application can use message channels to implement this capability (see [Ex_rep_chan: a Replication Manager channel example \(page 246\)](#) for information about a sample program that demonstrates this use of message channels). For a Base API application, it is possible to use the communications channels it 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.

2. Can I use replication to partition my environment across multiple sites?

Yes, you can create partial views to accomplish this. See [Replication views \(page 212\)](#) for more information.

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?

Replication Manager provides its own communications infrastructure for replication messages. You can create message channels to pass application-specific messages using this infrastructure (see [Using Replication Manager message channels \(page 236\)](#) for more information).

In a Base API application, 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 \(page 208\)](#) 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 \(page 200\)](#)). 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 `-n` option is not supported by `ex_rep_mgr`. That option specifies the number of nodes in the replication group. When you use the Replication Manager, this number is automatically determined for you.

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** Identifies the helper site used for joining the group.
- R** Identifies a remote peer to be used for joining the group. This peer is used for syncing purposes. See [Client-to-client synchronization \(page 224\)](#) for more information.
- 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 \(page 226\)](#) for more information.
- n** Specify the total number of sites in the replication group (`ex_rep_base` only).
- p** Set the election priority. See [Elections \(page 221\)](#) for more information.
- v** Indicates that additional informational and debugging output should be enabled.

A typical `ex_rep_mgr` session begins with a command such as the following, to start a master:

```
ex_rep_mgr -M -p 100 -h DIR1 -l localhost:30100
```

and several clients:

```
ex_rep_mgr -C -p 50 -h DIR2 -l localhost:30101 -r localhost:30100
ex_rep_mgr -C -p 10 -h DIR3 -l localhost:30102 -r localhost:30100
ex_rep_mgr -C -p 0 -h DIR4 -l localhost:30103 -r localhost:30100
```

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 send 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 \(page 208\)](#) 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.

Ex_rep_chan: a Replication Manager channel example

`Ex_rep_chan`, found in the `examples/c/ex_rep_chan` subdirectory of the Berkeley DB distribution, is a simple but complete demonstration of a replicated application that uses the Replication Manager feature of channels to perform write forwarding. The application is a mock stock ticker. Although similar to the `ex_rep_mgr` example program, this example differs in that it provides an example of using Replication Manager message channels. Any site accepts a command to write data to the database. If the site is a client, then, using the channels feature, the application forwards the request to the master site. If the site is a master then the request is automatically handled locally. You can read and write stock values at any site without needing to know what site is currently the master.

The set of supported commands can be viewed with either the **help** or the **?** command. Several commands work with key/data pairs where the key is a stock symbol and the data is its value.

The command to retrieve and print the current site's database contents is **print** or simply an empty input line. To read the contents of the master's database from any site use the **get key key ...** command. That command will forward the read request to the master if necessary and return the key/data pairs for all given keys.

There are two commands to put data into the database. Both commands take one or more key/data pairs, all of which are written into the database in a single transaction at the master site. The **put** command sends the data to the master site, and simply waits for a status response. The **put_sync** command sends the data to the master site, and uses a transaction token returned by the master to wait for the contents of that put to be available on the local site. This serves as a demonstration of the [read your writes](#) consistency feature.

The Makefile created by the standard UNIX configuration will build the `ex_rep_chan` example on most platforms. Enter "make `ex_rep_chan`" to build it.

The synopsis for the program is as follows:

```
ex_rep_chan -h home -l host:port [-MC] [-r host:port] [-R host:port] [-p
priority] [-v]
```

The options 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**
Identifies the helper site used for joining the group.
- R**
Identifies a remote peer to be used for joining the group. This peer is used for syncing purposes. See [Client-to-client synchronization \(page 224\)](#) for more information.
- p**
Set the election priority. See [Elections \(page 221\)](#) for more information.
- v**
Indicates that additional informational and debugging output should be enabled.

A typical `ex_rep_chan` session begins with a command such as the following, to start a master:

```
ex_rep_chan -M -h DIR1 -l localhost:30100
```

and several clients:

```
ex_rep_chan -C -h DIR2 -l localhost:30101 -r localhost:30100
ex_rep_chan -C -h DIR3 -l localhost:30102 -r localhost:30100
ex_rep_chan -C -h DIR4 -l localhost:30103 -r localhost:30100
```

Chapter 13. Distributed Transactions

Introduction

An application must use distributed transactions whenever it wants transactional semantics either across operations in multiple Berkeley DB environments (even if they are on the same machine) or across operations in Berkeley DB and some other database systems (for example, Oracle server). Berkeley DB provides support for distributed transactions using a two-phase commit protocol. In order to use the two-phase commit feature of Berkeley DB, an application must either implement its own global transaction manager or use an XA-compliant transaction manager such as Oracle Tuxedo (as Berkeley DB can act as an XA-compliant resource manager).

This chapter explains Berkeley DB's XA-compliant resource manager, which can be used in any X/Open distributed transaction processing system, and explains how to configure Oracle Tuxedo to use the Berkeley DB resource manager.

Berkeley DB XA Implementation

Berkeley DB provides support for distributed transactions using the two-phase commit protocol via its `DB_TXN->prepare()` interfaces. The `DB_TXN->prepare()` method performs the first phase of a two-phase commit, flushing the log to disk, and associating a global transaction ID with the underlying Berkeley DB transaction. This global transaction ID is used by the global transaction manager to identify the Berkeley DB transaction, and will be returned by the `DB_ENV->txn_recover()` method when it is called during recovery.

However, Berkeley DB does not perform distributed deadlock detection. Instead, when being used as an XA resource manager, Berkeley DB acquires all locks in a non-blocking mode. This results in pre-emptive abort of transactions that have the potential to deadlock. While this can lead to more transactions being aborted than is strictly necessary, it avoids system-wide hanging due to distributed deadlocks.

When using distributed transactions, there is no way to perform hot backups of multiple environments and guarantee that the backups are globally transaction-consistent across these multiple environments. If backups are desired, then all write transactions should be suspended; that is, active write transactions must be allowed to complete and no new write transactions should be begun. Once there are no active write transactions, the logs may be copied for backup purposes and the backup will be consistent across the multiple environments.

Building a Global Transaction Manager

Managing distributed transactions and using the two-phase commit protocol of Berkeley DB from an application requires the application to provide the functionality of a global transaction manager (GTM). The GTM is responsible for the following:

- Communicating with the multiple environments (potentially on separate systems).
- Managing the global transaction ID name space.

- Maintaining state information about each distributed transaction.
- Recovering from failures of individual environments.
- Recovering the global transaction state after failure of the global transaction manager.

Communicating with multiple Berkeley DB environments

Two-phase commit is required if a transaction spans operations in multiple Berkeley DB environments or across operations in Berkeley DB and some other database systems. If the environments reside on the same machine, the application can communicate with each environment through its own address space with no additional complexity. If the environments reside on separate machines, the application may use its own messaging capability, translating messages on the remote machine into calls into the Berkeley DB library (including the recovery calls).

Recovering from GTM failure

If the GTM fails, it must first recover its local state. Assuming the GTM uses Berkeley DB tables to maintain state, it should run the `db_recover` utility (or the `DB_RECOVER` option to `DB_ENV->open()`) upon startup. Once the GTM is back up and running, it needs to review all its outstanding global transactions, that is, all transactions that are recorded but not yet completed.

The global transactions that have not yet reached the prepare phase should be aborted. For each global transaction that has not yet prepared, the GTM should send a message to each participant telling it to abort its transaction.

Next the GTM should review its log to identify all participating environments that have transactions in the preparing, aborting, or committing states. For each such participant, the GTM should issue a `DB_ENV->txn_recover()` call. Upon receiving responses from each participant, the GTM must decide the fate of each transaction and issue appropriate calls. The correct behavior is defined as follows:

preparing

if all participating environments return the transaction in the list of prepared but not yet completed transactions, then the GTM should commit the transaction. If any participating environment fails to return the transaction in this list, then the GTM must issue an abort to all environments participating in that global transaction.

committing

the GTM should send a commit to any environment that returned this transaction in its list of prepared but not yet completed transactions.

aborting

the GTM should send an abort to any environment that returned this transaction in its list of prepared but not yet completed transactions.

Managing the Global Transaction ID (GID) name space

A global transaction is a transaction that spans multiple environments. Each global transaction must have a unique transaction ID. This unique ID is the global transaction ID (GID). In Berkeley DB, global transaction IDs must be represented by the character array, `DB_GID_SIZE`

(currently 128 bytes). It is the responsibility of the global transaction manager to assign GIDs, guarantee their uniqueness, and manage the mapping of local transactions to GID. That is, for each GID, the GTM should know which local transaction managers participated. The Berkeley DB logging system or a Berkeley DB table could be used to record this information.

Maintaining state for each distributed transaction.

In addition to knowing which local environments participate in each global transaction, the GTM must also know the state of each active global transaction. As soon as a transaction becomes distributed (that is, a second environment participates), the GTM must record the existence of the global transaction and all participants (whether this must reside on stable storage or not depends on the exact configuration of the system). As new environments participate, the GTM must keep this information up to date.

When the GTM is ready to begin commit processing, it should issue `DB_TXN->prepare()` calls to each participating environment, indicating the GID of the global transaction. Once all the participants have successfully prepared, then the GTM must record that the global transaction will be committed. This record should go to stable storage. Once written to stable storage, the GTM can send `DB_TXN->commit()` requests to each participating environment. Once all environments have successfully completed the commit, the GTM can either record the successful commit or can somehow "forget" the global transaction.

If an application uses nested transactions (that is, the parent parameter is non-NULL in a call to `DB_ENV->txn_begin()`) then, only the parent transaction should call `DB_TXN->prepare()`, not any of the child transactions.

Should any participant fail to prepare, then the GTM must abort the global transaction. The fact that the transaction is going to be aborted should be written to stable storage. Once written, the GTM can then issue `DB_TXN->abort()` requests to each environment. When all aborts have returned successfully, the GTM can either record the successful abort or "forget" the global transaction.

In summary, for each transaction, the GTM must maintain the following:

- A list of participating environments
- The current state of each transaction (pre-prepare, preparing, committing, aborting, done)

Recovering from the failure of a single environment

If a single environment fails, there is no need to bring down or recover other environments (the only exception to this is if all environments are managed in the same application address space and there is a risk that the failure of the environment corrupted other environments). Instead, once the failing environment comes back up, it should be recovered (that is, conventional recovery, via the `db_recover` utility or by specifying the `DB_RECOVER` flag to `DB_ENV->open()` should be run). If the `db_recover` utility is used, then the `-e` option must be specified. In this case, the application will almost certainly want to specify environmental parameters via a [DB_CONFIG configuration file \(page 136\)](#) in the environment's home directory, so that the `db_recover` utility can create an appropriately configured environment. If the `db_recover` utility is not used, then the GTM should call `DB_ENV->open()` specifying the `DB_RECOVER` flag. It should then call `DB_ENV->txn_recover()`, which will return an

array of DB_TXN handles for the set of prepared, but not yet completed transactions. For each transaction, the GTM should combine this knowledge with its transaction state table and call either DB_TXN->commit() or DB_TXN->abort(). After that process is complete, the environment is ready to participate in new transactions.

If the GTM is running in a system with multiple GTMs, it is possible that some of the transactions returned via DB_ENV->txn_recover() do not belong to the current environment. The GTM should detect this and call DB_TXN->discard() on each such transaction handle. Furthermore, it is important to note the environment does not retain information about which GTM has issued DB_ENV->txn_recover() operations. Therefore, each GTM should issue all its DB_ENV->txn_recover() calls before another GTM issues its calls. If the calls are interleaved, each GTM may not get a complete and consistent set of transactions. The simplest way to enforce this is for each GTM to make sure it can receive all its outstanding transactions in a single DB_ENV->txn_recover() call. The maximum number of possible outstanding transactions is roughly the maximum number of active transactions in the environment (whose value can be obtained using the db_stat utility). To simplify this procedure, the caller should allocate an array large enough to hold the list of transactions (for example, allocate an array able to hold three times the maximum number of transactions). If that is not possible, callers should check that the array was not completely filled in when DB_ENV->txn_recover() returns. If the array was completely filled in, each transaction should be explicitly discarded, and the call repeated with a larger array.

The newly recovered environment will forbid any new transactions from being started until the prepared but not yet completed transactions have been resolved. In the multiple GTM case, this means that all GTMs must recover before any GTM can begin issuing new transactions.

The GTM must determine how long it needs to retain global transaction commit and abort records. If the participating environments are following a DB_TXN_SYNC policy, that is, they are forcing commit and abort records to disk before replying to the GTM, then once the GTM has heard from all participants, it need not retain its persistent log records. However, if participating environments are running at weaker durability levels, such as DB_TXN_WRITE_NOSYNC or DB_TXN_NOSYNC, then the GTM must retain all commit and abort records until all participants have completed a checkpoint following the completion of a transaction.

Recovering from GTM failure

If the GTM fails, it must first recover its local state. Assuming the GTM uses Berkeley DB tables to maintain state, it should run the db_recover utility (or the DB_RECOVER option to DB_ENV->open()) upon startup. Once the GTM is back up and running, it needs to review all its outstanding global transactions, that is, all transactions that are recorded but not yet completed.

The global transactions that have not yet reached the prepare phase should be aborted. For each global transaction that has not yet prepared, the GTM should send a message to each participant telling it to abort its transaction.

Next the GTM should review its log to identify all participating environments that have transactions in the preparing, aborting, or committing states. For each such participant,

the GTM should issue a `DB_ENV->txn_recover()` call. Upon receiving responses from each participant, the GTM must decide the fate of each transaction and issue appropriate calls. The correct behavior is defined as follows:

preparing

if all participating environments return the transaction in the list of prepared but not yet completed transactions, then the GTM should commit the transaction. If any participating environment fails to return the transaction in this list, then the GTM must issue an abort to all environments participating in that global transaction.

committing

the GTM should send a commit to any environment that returned this transaction in its list of prepared but not yet completed transactions.

aborting

the GTM should send an abort to any environment that returned this transaction in its list of prepared but not yet completed transactions.

XA Introduction

Berkeley DB can be used as an XA-compliant resource manager. The XA implementation is known to work with the Tuxedo transaction manager.

The XA support is encapsulated in the resource manager switch `db_xa_switch`, which defines the following functions:

- `__db_xa_close`. Close the resource manager.
- `__db_xa_commit`. Commit the specified transaction.
- `__db_xa_complete`. Wait for asynchronous operations to complete.
- `__db_xa_end`. Disassociate the application from a transaction.
- `__db_xa_forget`. Forget about a transaction that was heuristically completed. (Berkeley DB does not support heuristic completion.)
- `__db_xa_open`. Open the resource manager.
- `__db_xa_prepare`. Prepare the specified transaction.
- `__db_xa_recover`. Return a list of prepared, but not yet committed transactions.
- `__db_xa_rollback`. Abort the specified transaction.
- `__db_xa_start`. Associate the application with a transaction.

The Berkeley DB resource manager does not support the following optional XA features:

- Asynchronous operations
- Transaction migration

The Tuxedo System is available from [Oracle BEA Systems](#).

Configuring Berkeley DB with the Tuxedo System

To configure the Tuxedo system to use Berkeley DB resource managers, do the following:

Update the Resource Manager File in Tuxedo

For the purposes of this discussion, assume that the Tuxedo home directory is in

```
/home/tuxedo
```

In that case, the resource manager file will be located in

```
/home/tuxedo/udataobj/RM
```

Edit the resource manager file to identify the Berkeley DB resource manager, the name of the resource manager switch, and the name of the library for the resource manager.

For example, on a RedHat Linux Enterprise (64-bit) installation of Oracle Tuxedo 11gR1, you can update the resource manager file by adding the following line:

```
BERKELEY-DB:db_xa_switch:-L${DB_INSTALL}/lib -ldb
```

where `${DB_INSTALL}` is the directory into which you installed the Berkeley DB library.

Note that the load options may differ depending on the platform of your system.

Build the Transaction Manager Server

To do this, use the Tuxedo **buildtms(1)** utility. The **buildtms** command will create the Berkeley-DB resource manager in the directory from which it was run. The parameters to the **buildtms** command should be:

```
buildtms -v -o DBRM -r BERKELEY-DB
```

This will create an executable transaction manager server, DBRM, which is called by Tuxedo to process begins, commits, and aborts.

Update the UBBCONFIG File

You must make sure that your TUXCONFIG environment variable identifies an UBBCONFIG file that properly identifies your resource managers. In the GROUPS section of the UBBCONFIG file, you should identify the group's LMID and GRPNO, as well as the transaction manager server name "TMSNAME=DBRM." You must also specify the OPENINFO parameter, setting it equal to the string

```
rm_name:dir
```

where `rm_name` is the resource name specified in the RM file (that is, BERKELEY-DB) and `dir` is the directory for the Berkeley DB home environment (see `DB_ENV->open()` for a discussion of Berkeley DB environments).

Because Tuxedo resource manager startup accepts only a single string for configuration, any environment customization that might have been done via the config parameter to `DB_ENV->open()` must instead be done by placing a [DB_CONFIG configuration file \(page 136\)](#) in the Berkeley DB environment directory. See [File naming \(page 136\)](#) for further information.

Consider the following configuration. We have built a transaction manager server, as described previously. We want the Berkeley DB environment to be `/home/dbhome`, our database files to be maintained in `/home/datafiles`, our log files to be maintained in `/home/log`, and we want a duplexed server.

The GROUPS section of the ubb file might look like the following:

```
group_tm LMID=myname GRPNO=1 TMSNAME=DBRM TMSCOUNT=2 \  
OPENINFO="BERKELEY-DB:/home/dbhome"
```

There would be a [DB_CONFIG configuration file \(page 136\)](#) in the directory `/home/dbhome` that contained the following two lines:

```
add_data_dir    /home/datafiles  
set_lg_dir      /home/log
```

Finally, the UBBCONFIG file must be translated into a binary version using Tuxedo's `tmloadcf(1)` utility, and then the pathname of that binary file must be specified as your TUXCONFIG environment variable.

At this point, your system is properly initialized to use the Berkeley DB resource manager.

See DB class for further information on accessing data files using XA.

Restrictions on XA Transactions

When you are using Berkeley DB for XA transactions, there are a few restrictions you should be aware of:

- Configure environment using the DB_CONFIG file

For most options, you must configure your environment via the DB_CONFIG file because an XA application or server cannot control the environment creation.

- Snapshot isolation must be configured for the entire environment.

Transactions managed by the Berkeley DB X/open compliant XA resource manager can be configured for transaction snapshots using either database open flags or the DB_CONFIG file. To configure using database open flags, open the XA managed database with the flag `DB_MULTIVERSION`. When using DB_CONFIG, include both of the following lines:

```
set_flags DB_MULTIVERSION  
set_flags DB_TXN_SNAPSHOT
```

Note that both methods will results in all transactions using transaction snapshots, there is no way to enable transaction snapshots in just a subset of XA managed transactions.

- No in-memory logging

Upon return from `xa_open`, Berkeley DB checks to ensure there is no in-memory logging. If in-memory logging is detected, a FAILURE message is returned to the application.

- No application-level child transactions

Berkeley DB verifies in the `xa_start` and `xa_end` calls that no XA transaction has a parent. If application-level child transactions are detected, a `FAILURE` message is returned to the application.

- All database-level operations, such as create, rename, and remove, must be performed in local BDB transactions, not distributed XA transactions

Berkeley DB checks that there is no XA transaction currently active during these operations, and if detected, a `FAILURE` message is returned to the application.

- Close cursors before a service invocation returns

Berkeley DB checks in the `xa_end` call that the `DB_TXN` has no active cursors open and if detected, a `FAILURE` message is returned to the application.

XA: Frequently Asked Questions

1. Is it possible to mix XA and non-XA transactions?

Yes. It is also possible for XA and non-XA transactions to coexist in the same Berkeley DB environment. To do this, specify the same environment to the non-XA `DB_ENV->open()` calls as was specified in the Tuxedo configuration file.

2. Does converting an application to run within XA change any of the already existing C/C++ API calls it does?

When converting an application to run under XA, the application's Berkeley DB calls are unchanged, with three exceptions:

- a. The application must specify the `DB_XA_CREATE` flag to the `db_create()` function.
- b. Unless the application is performing an operation for a non-XA transaction, the application should never explicitly call `DB_TXN->commit()`, `DB_TXN->abort()`, and `DB_ENV->txn_begin()`, and those calls should be replaced by calls into the Tuxedo transaction manager.
- c. Unless the application is performing an operation for a non-XA transaction, the application should specify a transaction argument of `NULL` to Berkeley DB methods taking transaction arguments (for example, `DB->put()` or `DB->cursor()`).

Otherwise, the application should be unchanged.

3. How does Berkeley DB recovery interact with recovery by the Tuxedo transaction manager?

Recovery is completed in two steps. First, each resource manager should recover its environment(s). This can be done via a program that calls `DB_ENV->open()` or by calling the `db_recover` utility. If using the `db_recover` utility, then the option should be specified so that the regions that are recovered persist after the utility exits. Any transactions that were prepared, but neither completed nor aborted, are restored to their prepared

state so that they may be aborted or committed via the Tuxedo recovery mechanisms. After each resource manager has recovered, then Tuxedo recovery may begin. Tuxedo will interact with each resource manager via the `__db_xa_recover` function which returns the list of prepared, but not yet completed transactions. It should issue a commit or abort for each one, and only after having completed each transaction will normal processing resume.

Finally, standard log file archival and catastrophic recovery procedures should occur independently of XA operation.

4. Does Berkeley DB provide multi-threaded support for XA transactions?

Yes. For information on how to build multi-threaded servers for XA transactions, see http://download.oracle.com/docs/cd/E13161_01/tuxedo/docs10gr3/pgc/pgthr.html. All databases used by servers should be opened with handles created with the `DB_XA_CREATE` flag in the `db_create()` method and must be opened in the `tpsvrinit` routine. Note that the `environment` parameter of the `db_create()` method must be assigned `NULL`. For more information on the `tpsvrinit` routine, see http://download.oracle.com/docs/cd/E13161_01/tuxedo/docs10gr3/pgc/pgthr.html.

Chapter 14. 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 section on log format changes in the Upgrading Berkeley DB installations chapter of the Berkeley DB Installation and Build Guide. 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 <code>prev_lsn</code> field, automatically included in every auto-generated log record, should be returned through this argument. The <code>prev_lsn</code> 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 \(page 267\)](#) 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 15. 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.

Globalization Support

Berkeley DB globalization support allows you to translate error and informational message text to the language of your choosing, and then use the translated text instead of the default English text. This section describes Berkeley DB's globalization support. Berkeley DB's error and informational message text is captured in the [Berkeley DB Message Reference Guide](#).

Message Format

By default, Berkeley DB messages are comprised of a message number followed by message text in English. For example:

```
BDB1001 illegal record number size
```

It is possible to build Berkeley DB with stripped messages. When messages are stripped, the message text is removed from the library, leaving behind only the message number. When building a stripped library, there is no message text available so localization will not work.

If localization is enabled, the translated message is substituted for the original message text.

Enable Globalization Support

To output messages in a language other than the default English, follow the steps below:

1. Provide an i18n component containing a localization function used to translate messages, and translation files that map existing messages to localized messages. The localization function can be added to the current Berkeley DB project, or as a dynamic library that is called at run time.
2. Add the name of the localization function as the prefix for "(msg)" when HAVE_LOCALIZATION is defined in build_unix/db_int.in on *nix, or in build_windows/db_int.h on Windows.

3. On *nix, specify -DHAVE_LOCALIZATION to CFLAGS. On Windows, specify /D HAVE_LOCALIZATION to the C/C++ Additional Options in the db project properties.
4. Within your application code, use DB_ENV->set_errcall() or DB->set_errcall() to print the messages.

Note that Berkeley DB supports only UTF-8 for its message text. If your localization requires UTF-16 Unicode, the UTF-16 characters must be converted to UTF-8 Unicode by your localization function. If necessary, the error reporting function you specify to DB_ENV->set_errcall() or DB->set_errcall() can be used to revert the UTF-8 Unicode back to the UTF-16 Unicode.

Localization Example

The following example walks you through providing localization support for a single Berkeley DB error message:

- Make the resource bundles. These provide the actual text translation:

es.txt:

```
es {
  BDB1002 illegal record number of 0 {"BDB1002 illegal record number of 0"}
}
```

de_CH.txt:

```
de_CH {
  BDB1002 illegal record number of 0 {"BDB1002 illegale Rekordzahl von 0"}
}
```

root.txt:

```
root {
  BDB1002 illegal record number of 0 {"BDB1002 illegal record number of 0"}
}
```

- Write and compile your localization functions: Note that the "es", "de_CH" and "root" tags are used as the locale name in ures_open().

Also notice that because ures_getStringByKey() returns UTF-16 Unicode, its output is converted to UTF-8 using ucnv_fromUChars().

```
UConverter *conv;
UResourceBundle *rhandle;
char *mbuf;

initialize() {
  /* Open ICU resource, specify the locale. */
  rhandle = ures_open(resource, "de_CH", &status);
  /* Open an ICU converter. */
  conv = ucnv_open("iso-8859-3", &status);
```

```

    mbuf = malloc(len * sizeof(char));
    memset(mbuf, 0, 100 * sizeof(char));
}

translate() {
    const UChar *wmsg;
    /* Get the translated message from the resource. */
    wmsg = ures_getStringByKey(rhandle, src, &len, &status);
    /* Convert UChar * to char. */
    len = ucnv_fromUChars(conv, wmsg, 100, , -1, &status);
}

close() {
    ucnv_close(conv);
    ures_close(rhandle);
    free(mbuf);
}

```

- Update db_int.h so that _(msg) is defined to use the translate() that we created in the previous step.

```

#ifdef HAVE_LOCALIZATION
#define _(msg) translate(msg)
#else
#define _(msg) msg
#endif

```

- Rebuild Berkeley DB, making sure to specify the HAVE_LOCALIZATION compile option.
- Specify the error callback.

```

dbp->set_errcall(dbp, print_err);

print_err() {
    const UChar *wmsg;
    len = ucnv_toUChars(conv, wmsg, 100, src, len, &status);
    u_stdout = u_finit(stdout, NULL, NULL);
    u_file_write((UChar *)wmsg, len, u_stdout);
}

```

The result of this is if you input an incorrect recno and reach the error 1002, the message "BDB1002 illegale Rekordzahl von 0" is output.

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 \(page 136\)](#). 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 \(page 43\)](#)

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->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 \(page 139\)](#).

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](#) 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.

Performance Event Monitoring

The Performance Event Monitoring feature uses Solaris DTrace or Linux SystemTap to "publish" interesting events as they occur inside of Berkeley DB. The operating system utilities **dtrace** or **stap** run scripts which select, analyze, and display events. There is no need to modify the application. Any application which uses that Berkeley DB library can be monitored. For more information about these instrumentation tools refer to the following pages:

DTrace

<http://www.oracle.com/technetwork/server-storage/solaris11/technologies/dtrace-1930301.html>

SystemTap

<http://sourceware.org/systemtap/>

Performance Event Monitoring is available for operating systems where DTrace or SystemTap supports static probe points in user applications, such as Solaris 10 and OpenSolaris, some versions of Linux, and Mac OS X 10.6 and later. By including `--enable-dtrace` to the configuration options, the resulting libraries will include probe points for these event categories:

- database operations: opening a database or cursor, get, put, delete.
- internal operations regarding disk allocation, transactions, concurrency control, and caching.
- the beginning and ending of waiting periods due to conflicting transactional consistency locks (for example, page locks), mutexes, or shared latches.
- timing dependent code paths which are expected to be infrequent. These could raise concerns if they were to happen too often.

These probe points are implemented as user-level statically defined traces (USDT's) for DTrace, and static userspace markers for SystemTap.

To monitor the statistics values as they are updated include the `--enable-perfmon-statistics` configuration option. This option generates probe points for updates to many of the counters whose values are displayed by the `db_stat` utility or returned by the various statistics functions. The "cache" example script uses a few of these probe points.

Performance Event Monitoring is intended to be suitable for production applications. Running Berkeley DB with DTrace or SystemTap support built in has little effect on execution speed until probes are enabled at runtime by the **dtrace** or **stap** programs.

The list of available events may be displayed by running 'make listprobes' after building the libdb-6.0 shared library.

Using the DTrace Provider

The DTrace probe provider for Berkeley DB is named 'bdb'. A simple dtrace command to monitor all dtrace-enabled Berkeley DB activity in the system is:

```
dtrace -Zn 'bdb*::: { printf("%s", probename); }'
```

DTrace requires elevated privileges in order to run. On Solaris you can avoid running as root by giving any users who need to run **dtrace** the `dtrace_proc` or `dtrace_user` privilege in `/etc/user_attr`.

DTrace works on both 32 and 64 bit applications. However, when tracing a 32-bit application on a 64-bit processor it might be necessary to pass a "32 bit" option to **dtrace**. Without this option the D language might use a pointer size of 8 bytes, which could cause pointer values (and structures containing them) to be processed incorrectly. Use the `-32` option on Solaris; on Mac OS X use `-arch i386`.

Using SystemTap

SystemTap looks up its static userspace markers in the library name specified in the **stap** script. A simple **stap** command to list the probes of the default Berkeley DB installation is:

```
stap -l 'process("/usr/local/BerkeleyDB.6.0/lib/libdb-6.0.so").mark("*")'
```

Berkeley DB supports SystemTap version 1.1 or later. Building with userspace marker support requires `sys/sdt.h`, which is often available in the package `systemtap-sdt-devel`. Running **stap** with userspace markers requires that the kernel have "utrace" support; see <http://sourceware.org/systemtap/wiki/utrace> for more information.

SystemTap needs elevated privileges in order to run. You can avoid running as root by adding the users who need to run **stap** to the group `stapdev`.

Example Scripts

Berkeley DB includes several example scripts, in both DTrace and SystemTap versions. The DTrace examples end with a `.d` suffix and are located in `util/dtrace`. The SystemTap examples have a `.stp` suffix and can be found in `util/systemtap`. The Berkeley DB shared library name, including any necessary path, is expected as the first parameter to the SystemTap examples.

apicalls

This script graphs the count of the main API calls. The result is grouped by thread of the target process.

apitimes

This script graphs the time spent in the main API calls, grouped by thread.

apitrace

This script displays the entry to and return from each of the main API calls.

cache

This script displays overall and per-file buffer cache statistics every *N* (default: 1) seconds for *M* (default: 60) intervals. It prints the number of cache hits, misses, and evictions for each file with any activity during the interval.

dbdefs

This contains DTrace-compatible declarations of Berkeley DB data structures returned from probe points. There is no Linux equivalent; SystemTap can obtain type information directly from the debugging symbols compiled into the libdb*-6.0.so shared library.

locktimes

This script graphs the time spent waiting for DB page locks. The result times in nanoseconds are grouped by filename, pgno, and lock_mode. The optional integer maxcount parameter directs the script to exit once that many page lock waits have been measured.

locktimesid

This is similar to the locktimes script above, except that it displays the 20 byte file identifier rather than the file name. This can be useful when there are several environments involved, or when database files are recreated during the monitoring period.

mutex

This script measures mutex wait periods, summarizing the results two ways.

- The first grouping is by mutex, mode (exclusive or shared), and thread id.
- The second grouping is by the mutex category and mode (exclusive or shared). The mutex categories are the MTX_XXX definitions in dbinc/mutex.h.

showerror

This script displays the application stack when the basic error routines are called. It provides additional information about an error, beyond the string sent to the diagnostic output.

These examples are designed to trace a single process. They run until interrupted, or the monitored process exits, or the limit as given in an optional 'maximum count' argument has been reached.

Performance Events Reference

The events are described below as if they were functions with ANSI C-style signatures. The values each event provides to DTrace or SystemTap are the arguments to the functions.

alloc

The alloc class covers the allocation of "on disk" database pages.

alloc-new (char *file, char *db, unsigned pgno, unsigned type, struct _db_page *pg, int ret);

An attempt to allocate a database page of type 'type' for database 'db' returned 'ret'. If the allocation succeeded then ret is 0, pgno is the location of the new page, and pg is the address of the new page. Details of the page can be extracted from the pg pointer.

alloc-free (char *file, char *db, unsigned pgno, unsigned ret);

An attempt to free the page 'pgno' of 'db' returned 'ret'. When successful the page is returned to the free list or the file is truncated.

alloc-btree_split (char *file, char *db, unsigned pgno, unsigned parent, unsigned level);

A btree split of pgno in db is being attempted. The parent page number and the level in the btree are also provided.

db

These DB API calls provide the name of the file and database being accessed. In-memory databases will have a NULL (0) file name address. The db name will be null unless subdatabases are in use.

db-open (char *file, char *db, unsigned flags, uint8_t *fileid);

The database or file name was opened. The 20 byte unique fileid can be used to keep track of databases as they are created and destroyed.

db-close (char *file, char *db, unsigned flags, uint8_t *fileid);

The database or file name was closed.

db-cursor (char *file, char *db, unsigned txnid, unsigned flags, uint8_t *fileid);

An attempt is being made to open a cursor on the database or file.

db-get (char *file, char *db, unsigned txnid, DBT *key, DBT *data, unsigned flags);

An attempt is being made to get data from a db.

db-put (char *file, char *db, unsigned txnid, DBT *key, DBT *data, unsigned flags);

An attempt is being made to put data to a db.

db-del (char *file, char *db, unsigned txnid, DBT *key, unsigned flags);

An attempt is being made to delete data from a db.

lock

The lock class monitors the transactional consistency locks: page, record, and database. It also monitors the non-transactional file handle locks.

lock-suspend (DBT *lock, db_lockmode_t lock_mode);

The thread is about to suspend itself because another locker already has a conflicting lock on object 'lock'. The lock DBT's data points to a __db_iloc structure, except for the atypical program which uses application specific locking.

lock-resume (DBT *lock, db_lockmode_t lock_mode);

The thread is awakening from a suspend.

lock-put (struct __sh_dbt *lock, unsigned flags);

The lock is being freed.

lock-put_reduce_count (struct __sh_dbt *lock, unsigned flags);

The lock would have been freed except that its refcount was greater than 1.

These lock counters are included by --enable-perfmon-statistics.

lock-deadlock (unsigned st_ndeadlocks, unsigned locker_id, struct __sh_dbt *lock_obj);

The locker_id's lock request in lock_obj is about to be aborted in order to resolve a deadlock. The lock region's st_ndeadlocks has been incremented.

lock-nowait_notgranted (unsigned count, DBT *lock, unsigned locker_id);

A DB_LOCK_NOWAIT lock request by locker_id would have had to wait. The lock regions's st_lock_nowait has been incremented and the request returns DB_LOCK_NOTGRANTED.

lock-steal (unsigned st_locksteals, unsigned from, unsigned to);

A lock is being stolen from one partition for another one. The 'from' lock partition's st_locksteals has been incremented.

lock-object_steal (unsigned st_objectsteals, unsigned from, unsigned to);

A lock object is being stolen from one partition for another one. The 'from' lock partition's st_objectsteals has been incremented.

lock-locktimeout (unsigned st_nlocktimeouts, const DBT *lock);

A lock wait expired due to the lock request timeout.

lock-txntimeout (unsigned st_ntxntimeouts, const DBT *lock);

A lock wait expired due to the transaction's timeout.

lock-nlockers (unsigned active, unsigned locker_id);

The allocation or deallocation of the locker id changed the number of active locker identifiers.

lock-maxnlockers (unsigned new_max_active, unsigned locker_id);

The allocation of the locker id set a new maximum number of active locker identifiers.

mpool

The mpool class monitors the allocation and management of memory, including the cache.

mpool-read (char *file, unsigned pgno, struct __bh *buf);

Read a page from file into buf.

mpool-write (char *file, unsigned pgno, struct __bh *buf);

Write a page from buf to file.

mpool-env_alloc (unsigned size, unsigned region_id, unsigned reg_type);

This is an attempt to allocate size bytes from region_id. The reg_type is one of the reg_type_t enum values.

mpool-evict (char *file, unsigned pgno, struct __bh *buf);

The page is about to be removed from the cache.

mpool-alloc_wrap (unsigned alloc_len, int region_id, int wrap_count, int put_counter);

The memory allocator has incremented wrap_count after searching through the entire region without being able to fulfill the request for alloc_len bytes. As wrap_count increases the library makes more effort to allocate space.

These mpool counters are included by --enable-perfmon-statistics.

mpool-clean_eviction (unsigned st_ro_evict, unsigned region_id);

The eviction of a clean page from a cache incremented st_ro_evict.

mpool-dirty_eviction (unsigned st_rw_evict, unsigned region_id);

The eviction of a dirty page from a cache incremented st_rw_evict. The page has already been written out.

mpool-fail (unsigned failure_count, unsigned alloc_len, unsigned region_id);

An attempt to allocate memory from region_id failed.

mpool-hash_search (unsigned st_hash_searches, char *file, unsigned pgno);

A search for pgno of file incremented st_hash_searches.

mpool-hash_examined (unsigned st_hash_examined, char *file, unsigned pgno);

A search for pgno of file increased st_hash_examined by the number of hash buckets examined.

mpool-hash_longest (unsigned st_hash_longest, char *file, unsigned pgno);

A search for pgno of file set a new maximum st_hash_longest value.

mpool-map (unsigned st_map, char *file, unsigned pgno);

A file's st_map count was incremented after a page was mapped into memory. The mapping might have caused disk I/O.

mpool-hit (unsigned st_cache_hit, char *file, unsigned pgno);

The hit count was incremented because pgno from file was found in the cache.

mpool-miss (unsigned st_cache_miss, char *file, unsigned pgno);

The miss count was incremented because pgno from file was not already present in the cache.

mpool-page_create (unsigned st_page_create, char *file, unsigned pgno);

The st_page_create field was incremented because the pgno of file was created in the cache.

mpool-page_in (unsigned st_page_in, char *file, unsigned pgno);

The st_page_in field was incremented because the pgno from file was read into the cache.

mpool-page_out (unsigned st_page_out, char *file, unsigned pgno);

The st_page_out field was incremented because the pgno from file was written out.

mutex

The mutex category monitors includes shared latches. The alloc_id value is one of the MTX_XXX definitions from dbinc/mutex.h

mutex-suspend (unsigned mutex, unsigned excl, unsigned alloc_id, struct __db_mutex_t *mutexp);

This thread is about to suspend itself because a thread has the mutex or shared latch locked in a mode which conflicts with the this request.

mutex-resume (unsigned mutex, unsigned excl, unsigned alloc_id, struct __db_mutex_t *mutexp);

The thread is returning from a suspend and will attempt to obtain the mutex or shared latch again. It might need to suspend again.

These mutex counters are included by --enable-perfmon-statistics.

mutex-set_nowait (unsigned mutex_set_nowait, unsigned mutex);

Increment the count of times that the mutex was free when trying to lock it.

mutex-set_wait (unsigned mutex_set_wait, unsigned mutex);

Increment the count of times that the mutex was busy when trying to lock it.

mutex-set_rd_nowait (unsigned mutex_set_rd_nowait, unsigned mutex);

Increment the count of times that the shared latch was free when trying to get a shared lock on it.

mutex-set_rd_wait (unsigned mutex_set_rd_wait, unsigned mutex);

Increment the count of times that the shared latch was already exclusively latched when trying to get a shared lock on it.

mutex-hybrid_wait (unsigned hybrid_wait, unsigned mutex);

Increment the count of times that a hybrid mutex had to block on its condition variable. In a busy system this might happen several times before the corresponding hybrid_wakeup.

mutex-hybrid_wakeup (unsigned hybrid_wakeup, unsigned mutex);

Increment the count of times that a hybrid mutex finished one or more waits for its condition variable.

txn

The txn category covers the basic transaction operations.

txn-begin (unsigned txnid, unsigned flags);

A transaction was successfully begun.

txn-commit (unsigned txnid, unsigned flags);

A transaction is starting to commit.

txn-prepare (unsigned txnid, uint8_t *gid);

The transaction is starting to prepare, flushing the log so that a future commit can be guaranteed to succeed. The global identifier field is 128 bytes long.

txn-abort (unsigned txnid);

The transaction is about to abort.

These txn counters are included by --enable-perfmon-statistics.

txn-nbegins (unsigned st_nbegins, unsigned txnid);

Beginning the transaction incremented st_nbegins.

txn-naborts (unsigned st_nbegins, unsigned txnid);

Aborting the transaction incremented st_naborts.

txn-ncommits (unsigned st_ncommits, unsigned txnid);

Committing the transaction incremented st_ncommits.

txn-nactive (unsigned st_nactive, unsigned txnid);

Beginning or ending the transaction updated the number of active transactions.

txn-maxnactive (unsigned st_maxnactive, unsigned txnid);

The creation of the transaction set a new maximum number of active transactions.

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.

3. How do I handle a "pass 4" error when trying to run one of the example Performance Event Monitoring scripts on my Linux system? The library was configured with --enable-dtrace and built without error.

A Linux installation can have SystemTap support for kernel probe points without including the kernel "utrace" module needed to use userspace probes. Pass 4 errors can occur when this required userspace support is not present.

4. I have a program with multiple threads running on multi-core systems, and it uses a shared DB handle for all the threads. The program does not behave as well as I expect and I see a lot of contention on the dbp->mutex, what should I do to reduce this contention ?

When running multi-threaded program on a multi-core/multi-processor system, we suggest using a DB handle per thread instead of using a shared handle accorss threads. Since many operations(like creating cursors) requires to lock the DB handle first, and if using a shared handle, there could be a lot of lock contention on the handle.

Chapter 16. 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_get()` 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 \(page 298\)](#) 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 267), 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.

For more information on the locking subsystem methods, see the Locking Subsystem and Related Methods section in the *Berkeley DB C API Reference Guide*.

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 amount of memory available to the locking system is specified using the `DB_ENV->set_memory_max()` method. Sizing of the environment using the `DB_ENV->set_memory_max()`

method is discussed in [Sizing a database environment \(page 132\)](#). Here we will discuss how to estimate the number of objects your application is likely to lock. Since running out of memory for locking structures is a fatal error requiring reconfiguration and restarting the environment it is best to overestimate the numbers.

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 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 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 number of lockers can be estimated as follows:

- If the database environment is using transactions, the number of lockers can be estimated by adding the number of simultaneously active non-transactional cursors and 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 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 number of lock objects needed for a transaction can be estimated as follows:

- For each access to a non-Queue database, one lock object is needed for each page that is read or updated.
- For the Queue access method you will need one lock object per record that is read or updated. Deleted records skipped by a DB_NEXT or DB_PREV operation do not require a separate lock object.
- For Btree and Recno databases additional lock objects may be needed for each node in the btree that has to be split due to an update.
- For Hash and Queue databases, every access must obtain a lock on the metadata page for the duration of the access. This is not held to the end of the transaction.
- If the transaction performs an update that needs to allocate a page to the database then a lock object for the metadata page will be needed to the end of the transaction.

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. So if a transaction in your application typically accesses

10 records, that transaction will require about 10 lock objects (it may be a few more if it splits btree nodes). If you have up to 10 concurrent threads in your application, then you need to configure your system to have about 100 lock objects. It is always better to configure more than you need so that you don't run out of lock objects. The memory overhead of over-allocating lock objects is minimal as they are small structures.

The number of locks required by an application cannot be easily estimated. It is possible to calculate a number of locks by multiplying the number of lockers, times the 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.

By default a minimum number of locking objects are allocated at startup. To avoid contention due to allocation the application may use the `DB_ENV->set_memory_init()` method to preallocate and initialize the following lock structures:

- `DB_MEM_LOCK`

Specifies the number of locks that can be simultaneously requested in the system.

- `DB_MEM_LOCKER`

Specifies the number of lockers that can simultaneously request locks in the system.

- `DB_MEM_LOCKOBJECTS`

Specifies the number of objects that can simultaneously be locked in the system.

In addition to the above structures, sizing your locking subsystem also requires specifying the number of lock table partitions. You do this using the `DB_ENV->set_lk_partitions()` method. Each partition may be accessed independently by a thread. 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 because the locking system is more efficient when there is a single partition. Some operating systems (Linux, Solaris) may report thread contexts as cpus, and so 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 is 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. The final values specified for the locks and lock objects should be more than or equal to the number of lock table partitions.

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 \(page 288\)](#) 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 \(page 267\)](#) 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 \(page 267\)](#) 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 \(page 267\)](#) 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 \(page 267\)](#) 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 \(page 267\)](#) 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 called 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 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 17. 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 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.

For more information on the logging subsystem methods, see the Logging Subsystem and Related Methods section in the *Berkeley DB C API Reference Guide*.

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 \(page 303\)](#) 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:

$$(10 * 2^{20} * 2000000000) / (6000 * 500 * 365 * 60 * 60 * 24) = \sim 221$$

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 \(page 181\)](#) 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 18. 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.

For more information on the memory pool subsystem methods, see the *Memory Pools and Related Methods* section in the *Berkeley DB C API Reference Guide*.

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 \(page 24\)](#).

Note the memory pool defaults to assuming that the average page size is 4k. This factor is used to determine the size of the hash table used to locate pages in the memory pool. The size of the hash table is calculated to so that on average 2.5 pages will be in each hash table entry. Each page requires a mutex be allocated to it and the average page size is used to determine the number of mutexes to allocate to the memory pool.

Normally you should see good results by using the default values for the page size, but in some cases you may be able to achieve better performance by manually configuring the page size. The expected page size, hash table size and mutex count can be set via the methods: `DB_ENV->set_mp_pagesize()`, `DB_ENV->set_mp_tablesize()`, and `DB_ENV->set_mp_mtxcount()`.

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.

Warming the memory pool

Some applications find it is useful to pre-load the memory pool upon application startup. This is a strictly optional activity that provides faster initial access to your data at the expense of longer application startup times.

To warm the cache, you simply have to read the records that your application will operate on most frequently. You can do this with normal database reads, and you can also use cursors. But the most efficient way to warm the cache is to use memory pool APIs to get the pages that contain your most frequently accessed records.

You read pages into the memory pool using the `DB_MPPOOLFILE->get()` method. This method acquires locks on the page, so immediately upon getting the page you need to put it so as to release the locks.

Also, you obtain a memory pool file handle using a database handle. This means that if your data is contained in more than one Berkeley DB database, you must operate on each database handle in turn.

The following example code illustrates this. It does the following:

- Opens an environment and two database handles.
- Determines how many database pages can fit into the memory pool.
- Uses `DB_MPPOOLFILE->get()` and `DB_MPPOOLFILE->put()` to load that number of pages into the memory pool.

First, we include the libraries that we need, forward declare some functions, and initialize some variables.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <db.h>

/* Forward declarations */
int warm_cache(DB *, int *, int);
int open_db(DB_ENV *, DB **, const char *);

int
main(void)
{
    DB *dbp1 = 0, *dbp2 = 0;
    DB_ENV *envp = 0;
    u_int32_t env_flags, pagesize, gbytes, bytes;
    int ret = 0, ret_t = 0, numcachepages, pagecount;
```

Then we open the environment and our databases. The `open_db()` function that we use here simply opens a database. We will provide that code at the end of this example, but it should hold no surprises for you. We only use the function so as to reuse the code.

```

/*
 * Open the environment and the databases
 */
ret = db_env_create(&envp, 0);
if (ret != 0) {
    fprintf(stderr, "Error creating environment handle: %s\n",
        db_strerror(ret));
    goto err;
}

env_flags =
    DB_CREATE      | /* Create the environment if it does
                       not exist */
    DB_RECOVER     | /* Run normal recovery. */
    DB_INIT_LOCK   | /* Initialize the locking subsystem */
    DB_INIT_LOG    | /* Initialize the logging subsystem */
    DB_INIT_TXN    | /* Initialize the transactional subsystem. This
                       * also turns on logging. */
    DB_INIT_MPOOL; /* Initialize the memory pool */

/* Now actually open the environment */
ret = envp->open(envp, "./env", env_flags, 0);
if (ret != 0) {
    fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
    goto err;
}

ret = open_db(envp, &dbp1, "mydb1.db");
if (ret != 0)
    goto err;

ret = open_db(envp, &dbp2, "mydb2.db");
if (ret != 0)
    goto err;

```

Next we determine how many database pages we can fit into the cache. We do this by finding out how large our pages are, and then finding out how large our cache can be.

```

/* Find out how many pages can fit at most in the cache */
ret = envp->get_mp_pagesize(envp, &pagesize);
if (ret != 0) {
    fprintf(stderr, "Error retrieving the cache pagesize: %s\n",
        db_strerror(ret));
    goto err;
}

ret = envp->get_cache_max(envp, &gbytes, &bytes);
if (ret != 0) {
    fprintf(stderr, "Error retrieving maximum cache size: %s\n",

```

```

        db_strerror(ret));
        goto err;
    }
    /* Avoid an overflow by first calculating pages per gigabyte. */
    numcachepages = gbytes * ((1024 * 1024 * 1024) / pagesize);
    numcachepages += bytes / pagesize;

```

Now we call our `warm_cache()` function. We will describe this function in a little while, but note that we call `warm_cache()` twice. This is because our example uses two databases, and the memory pool methods operate on a per-handle basis.

```

/*
 * Warm the cache by loading pages from each of the databases
 * in turn.
 */
pagecount = 0;
ret = warm_cache(dbp1, &pagecount, numcachepages);
if (ret != 0) {
    fprintf(stderr, "Error warming the cache: %s\n",
        db_strerror(ret));
    goto err;
}

ret = warm_cache(dbp2, &pagecount, numcachepages);
if (ret != 0) {
    fprintf(stderr, "Error warming the cache: %s\n",
        db_strerror(ret));
    goto err;
}

```

Now we close all our handles and finish our `main()` function. Again, this is straight-forward boilerplate code that we provide simply to be complete.

```

err:
    /* Close our database handles, if they were opened. */
    if (dbp1 != NULL) {
        ret_t = dbp1->close(dbp1, 0);
        if (ret_t != 0) {
            fprintf(stderr, "dbp1 close failed: %s\n",
                db_strerror(ret_t));
            ret = ret_t;
        }
    }

    if (dbp2 != NULL) {
        ret_t = dbp2->close(dbp2, 0);
        if (ret_t != 0) {
            fprintf(stderr, "dbp2 close failed: %s\n",
                db_strerror(ret_t));
            ret = ret_t;
        }
    }

```

```

    }

    /* Close our environment, if it was opened. */
    if (envp != NULL) {
        ret_t = envp->close(envp, 0);
        if (ret_t != 0) {
            fprintf(stderr, "environment close failed: %s\n",
                db_strerror(ret_t));
            ret = ret_t;
        }
    }

    /* Final status message and return. */
    printf("I'm all done.\n");
    return (ret == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}

```

As noted above, this example uses an `open_db()` function, which opens a database handle inside the provided environment. To be complete, this is the implementation of that function:

```

/* Open a database handle */
int
open_db(DB_ENV *envp, DB **dbpp, const char *file_name)
{
    int ret = 0;
    u_int32_t db_flags = 0;
    DB *dbp;

    /* Open the database */
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening database: %s : %s\n",
            file_name, db_strerror(ret));
        return ret;
    }

    /* Point to the memory malloc'd by db_create() */
    *dbpp = dbp;

    db_flags = DB_CREATE          | /* Create the database if it does
                                   not exist */
              DB_AUTO_COMMIT;     /* Allow autocommit */

    ret = dbp->open(dbp,          /* Pointer to the database */
                   0,             /* Txn pointer */
                   file_name,     /* File name */
                   0,             /* 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 open failed: %s : %s\n",
            file_name, db_strerror(ret));
        return ret;
    }
    return 0;
}

```

The warm_cache() function

In this section we provide the implementation of the warm_cache() function. This example function simply loads all the database pages that will fit into the memory pool. It starts from the first database page and continues until it either runs out of database pages or it runs out of room in the memory pool.

```

/* Warm the cache */
int
warm_cache(DB *dbp, int *pagecountp, int numcachepages)
{
    DB_MPOOLFILE *mpf = 0;
    void *page_addrp = 0;
    db_pgno_t page_number = 0;
    int ret = 0;
    int pagecount = *pagecountp;

    /*
     * Get the mpool handle
     */
    mpf = dbp->get_mpf(dbp);

    /* Load pages until there are no more pages in the database,
     * or until we've put as many pages into the cache as will fit.
     */
    while (ret != DB_PAGE_NOTFOUND && pagecount < numcachepages) {
        /*
         * Get the page from the cache. This causes DB to retrieve
         * the page from disk if it isn't already in the cache.
         */
        ret = mpf->get(mpf, &page_number, 0, 0, &page_addrp);
        if (ret && ret != DB_PAGE_NOTFOUND) {
            fprintf(stderr, "Error retrieving db page: %i : %s\n",
                page_number, db_strerror(ret));
            return ret;
        }

        /*
         * If a page was retrieved, put it back into the cache. This
         * releases the page latch so that the page can be evicted
         * if DB needs more room in the cache at some later time.
         */
    }
}

```

```
        if (ret != DB_PAGE_NOTFOUND) {
            ret = mpf->put(mpf, page_addrp, DB_PRIORITY_UNCHANGED, 0);
            if (ret) {
                fprintf(stderr, "Error putting db page: %i : %s\n",
                    page_number, db_strerror(ret));
                return ret;
            }
        }
        ++page_number;
        ++pagecount;
        *pagecountp = pagecount;
    }

    return 0;
}
```

Chapter 19. 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.

For more information on the transaction subsystem methods, see the Transaction Subsystem and Related Methods section in the *Berkeley DB C API Reference Guide*.

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 20. 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.

For more information on the operations supported by the sequence handle, see the Sequences and Related Methods section in the *Berkeley DB C API Reference Guide*.

Chapter 21. Berkeley DB Extensions: Tcl

Loading Berkeley DB with Tcl

Berkeley DB includes a dynamically loadable Tcl API, which requires that Tcl/Tk 8.5 or later already be installed on your system. You can download a copy of Tcl from the [Tcl Developer Xchange](#) 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](#) or [Building the Tcl API in the Berkeley DB Installation and Build Guide](#) 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:

```
# tclsh8.5
% lappend auto_path /usr/local/BerkeleyDB.6.0/lib
% pkg_mkIndex /usr/local/BerkeleyDB.6.0/lib libdb_tcl-6.0.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.6.0/lib/libdb_tcl-6.0.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.6.0/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. Additionally, the heap access method is not available.

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:

```
% berkbd 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 the Changing compile or load options section in the Berkeley DB Installation and Build Guide 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 8.5 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 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 22. 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 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.

For installation instructions, see the `INSTALL` file that resides in the `lang/php_db4` directory in your Berkeley DB distribution.

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_INIT_TXN,
              $mode = 0666) {}
function remove($home, $flags = 0) {}
function add_data_dir($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, $flags = 0) {} // returns statistics as
                                             // an array
    function join($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.

Chapter 23. 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 is 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 24. 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](#))

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](#))

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](#))

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](#))

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](#), 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>.