Oracle Berkeley DB

Getting Started with the SQL APIs

12c Release 1

Library Version 12.1.6.2

ORACLE

BERKELEY DB

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 4/13/2017

Table of Contents

Preface

Welcome to the Berkeley DB SQL interface. This manual describes how to configure and use the SQL interface to Berkeley DB 12*c* Release 1. This manual also describes common administrative tasks, such as backup and restore, database dump and load, and data migration when using the BDB SQL interface.

This manual is intended for anyone who wants to use the BDB SQL interface. Because usage of the BDB SQL interface is very nearly identical to SQLite, prior knowledge of SQLite is assumed by this manual. No prior knowledge of Berkeley DB is necessary, but it is helpful.

To learn about SQLite, see the official SQLite website at: http://www.sqlite.org

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Keywords or literal text that you are expected to type is presented in a `monospaced font`. For example: "Use the `DB_HOME` environment variable to identify the location of your environment directory."

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

Program examples and literal text that you might type 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 using the Berkeley DB SQL interface:

- Berkeley DB Installation and Build Guide
- Berkeley DB Programmer's Reference Guide

- Berkeley DB Getting Started with Replicated Applications

To download the latest 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 downloads, visit http://www.oracle.com/technetwork/database/database-technologies/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. Berkeley DB SQL: The Absolute Basics

Welcome to the Berkeley DB SQL interface. If you are a SQLite user who is using the BDB SQL interface for reasons other than performance enhancements, this chapter tells you the minimum things you need to know about the interface. You should simply read this chapter and then skip the rest of this book.

If, however, you are using the BDB SQL interface for performance reasons, then you need to read this chapter, plus most of the rest of the chapters in this book (although you can probably skip most of Administrating Berkeley DB SQL Databases (page 42), unless you want to administer your database "the Berkeley DB way").

Also, if you are an existing Berkeley DB user who is interested in the BDB SQL interface, read this chapter plus the rest of this book.

BDB SQL Is Nearly Identical to SQLite

Your interaction with the BDB SQL interface is almost identical to SQLite. You use the same APIs, the same command shell environment, the same SQL statements, and the same PRAGMAs to work with the database created by the BDB SQL interface as you would if you were using SQLite.

To learn how to use SQLite, see the official SQLite Documentation Page.

That said, there are a few small differences between the two interfaces. These are described in the remainder of this chapter.

Getting and Installing BDB SQL

The BDB SQL interface comes as a part of the Oracle Berkeley DB download. This can be downloaded from the Oracle Berkeley DB download page.

On Windows Systems

The BDB SQL interface is automatically built and installed whenever you build or install Berkeley DB for a Windows system. The BDB SQL interface `dll`s and the command line interpreter have names that differ from a standard SQLite distribution as follows:

- `dbsql.exe`

  This is the command line shell. It operates identically to the SQLite **sqlite3.exe** shell.

- `libdb_sql60.dll`

  This is the library that provides the BDB SQL interface. It is the equivalent of the SQLite `sqlite3.dll` library.

Note

If you are upgrading an existing BDB SQL installation, and you are upgrading from release 6.1.19 or lower, then see the SQL database upgrade instructions at Updated SQLite Version in the Berkeley DB Installation and Build Guide.

On Unix

In order to build the BDB SQL interface, you download and build Berkeley DB, configuring it so that the BDB SQL interface is also built. Be aware that it is not built by default. Instead, you need to tell the Berkeley DB `configure` script to also build the BDB SQL interface. For instructions on building the BDB SQL interface, see Building the DB SQL Interface in the *Berkeley DB Installation and Build Guide*.

The library and application names used when building the BDB SQL interface are different than those used by SQLite. If you want library and command shell names that are consistent with the names used by SQLite, configure the BDB SQL interface build using the compatibility (`--enable-sql_compat`) option.

Warning

The compatibility option can break other applications on your platform that rely on standard SQLite. This is especially true of Mac OS X, which uses standard SQLite for a number of default applications.

Use the compatibility option only if you know exactly what you are doing.

Unless you built the BDB SQL interface with the compatibility option, libraries and a command line shell are built with the following names:

- dbsql

  This is the command line shell. It operates identically to the SQLite **sqlite3** shell.

- `libdb_sql`

  This is the library that provides the BDB SQL interface. It is the equivalent of the SQLite `libsqlite3` library.

The BDB SQL ADO.NET Interface

Download the ADO.NET package from the Oracle Berkeley DB download page.

Prerequisites For Building The ADO.NET Package

- To build the Linq package, you will need to install `Microsoft .NET Framework 3.5 SP1`.
- To build SQLite.Designer, you will need to install the `Microsoft Visual Studio SDK`.
- To build on Windows Mobile you will need to install the `Microsoft Windows Mobile 6.5.3 Developer Tool Kit (DTK)`.
- To build on Windows Mobile you will need to use Visual Studio 2008.

Building BDB SQL ADO.NET Interface For Windows

- The package contains Visual Studio solution files:

- `SQLite.NET.2008.sln` and `SQLite.NET.2010.sln`

  For use by with Visual Studio 2008 or 2010. Note that these solution files do not build support for Linq or SQLite Designer.

- `SQLite.NET.2008.MSBuild.sln` and `SQLite.NET.2010.MSBuild.sln`

  For use with MSBuild (Microsoft Build Engine). These can also be used with Visual Studio. These solutions exclude SQLite Designer and CompactFramework. By default, these do not build support for Linq.

- Change the current platform target to `ReleaseNativeOnly` choose either `Win32` or `x64` depending on your target platform.
- Build the solution.

Building BDB SQL ADO.NET Interface For Windows Mobile

Building BDB SQL ADO.NET for Windows Mobile requires Windows Mobile 6.5.3 Professional DTK. Typical requirements for installing this toolkit are:

- Visual Studio 2005 SP1 or Later
- ActiveSync 4.5
- .NET CompactFramework 2.0 SP1
- Windows Mobile 6 SDK

To build BDB SQL ADO.NET for Windows Mobile, do the following:

- Open the `SQLite.NET.2008.WinCE.sln` solution file in Visual Studio 2008.
- Select `Load Project Normally`
- Change the current platform to `ReleaseNativeOnly`.
- Select `Configuration Manager->new`, then type or select the platform `Windows Mobile 6.5.3 Professional DTK (ARMV4I)`. Choose to copy settings from `Pocket PC 2003 (ARMV4I)`
- Build the solution.

Accessing BDB SQL Databases

BDB SQL databases can be accessed using a number of different drivers, applications and APIs. Only some of these are supported by all major platforms, as identified in the following table.

	UNIX/POSIX	Windows	Windows Mobile/CE	Android	iOS
DBSQL Library	x	x	x	x	x

	UNIX/POSIX	Windows	Windows Mobile/CE	Android	iOS
DBSQL Shell	x	x	x	x	x
ODBC	x	x			
JDBC	x	x			
ADO.NET		x	x		

The Journal Directory

When you create a database using the BDB SQL interface, a directory is created alongside of it. This directory has the same name as your database file, but with a `-journal` suffix.

That is, if you create a database called "mydb" then the BDB SQL interface also creates a directory alongside of the "mydb" file called "mydb-journal".

This directory contains files that are very important for the proper functioning of the BDB SQL interface. Do not delete this directory or any of its files unless you know what you are doing.

In Berkeley DB terms, the journal directory contains the environment files that are required to provide access to databases across multiple processes.

User Authentication

See the following section for more information:

- BDB User Authentication (page 4)
- BDB SQL Key-store Based User Authentication (page 7)

BDB User Authentication

With the BDB user authentication extension, a database can be marked as requiring authentication. To visit an authentication-required BDB database, an authenticated user must be logged into the database connection first. Once a BDB database is marked as authentication-required, it cannot be converted back into a no-authentication-required database. Encryption is mandatory if user authentication is activated.

By default a database does not require authentication. The BDB user authentication module will be activated by adding the -DBDBSQL_USER_AUTHENTICATION compile-time option. The client application must add `lang/sql/generated/sqlite3.h` to work with BDB user authentication. BDB user authentication is based on SQLite User Authentication.

See the following sections for more information:

- The Interface (page 5)
- Bootstrap (page 6)
- Transaction (page 6)

- Security Considerations (page 6)

The Interface

The users can use the following 3 ways to work with BDB user authentication:

- via C APIs

```
int sqlite3_user_authenticate(
  sqlite3 *db,           /* The database connection */
  const char *zUsername, /* Username */
  const char *aPW,       /* Password or credentials */
  int nPW                /* Number of bytes in aPW[] */
);
```

```
int sqlite3_user_add(
  sqlite3 *db,           /* Database connection */
  const char *zUsername, /* Username to be added */
  const char *aPW,       /* Password or credentials */
  int nPW,               /* Number of bytes in aPW[] */
  int isAdmin            /* True to give new user admin privilege */
);
```

```
int sqlite3_user_change(
  sqlite3 *db,           /* Database connection */
  const char *zUsername, /* Username to change */
  const void *aPW,       /* Modified password or credentials */
  int nPW,               /* Number of bytes in aPW[] */
  int isAdmin            /* Modified admin privilege for the user */
);
```

```
int sqlite3_user_delete(
  sqlite3 *db,           /* Database connection */
  const char *zUsername  /* Username to remove */
);
```

- Via the BDB SQL user authentication PRAGMAs below:
  - `PRAGMA bdbsql_user_login="{USER_NAME}:{USER_PWD}";`
  - `PRAGMA bdbsql_user_add="{USER_NAME}:{USER_PWD}:{IS_ADMIN}";`
  - `PRAGMA bdbsql_user_edit="{USER_NAME}:{USER_PWD}:{IS_ADMIN}";`
  - `PRAGMA bdbsql_user_delete="{USER_NAME}";`
- Via the BDB SQL shell commands as below:
  - `.user login {USER_NAME} {USER_PWD}`
  - `.user add {USER_NAME} {USER_PWD} {IS_ADMIN}`
  - `.user edit {USER_NAME} {USER_PWD} {IS_ADMIN}`

- `.user delete {USER_NAME}`

You can use the sqlite3_user_authenticate() interface to log in a user into the database connection. Calling sqlite3_user_authenticate() on a no-authentication-required database connection will return an error. This is different from the original SQLite behavior.

You can use the sqlite3_user_add()/sqlite3_user_delete() interfaces to add/delete a user. It resuts in an error to call sqlite3_user_add()/sqlite3_user_delete() on an authentication-required database connection without an administrative user loged in. The currently logged-in user cannot be deleted.

You can use the sqlite3_user_change() interface to change a users login credentials or admin privilege. Any user can change their own password, but no user can change their own administrative privilege setting. Only an administrative user can change another users login credentials or administrative privilege setting.

The sqlite3_set_authorizer() callback is modified to take a 7th parameter which is the username of the currently logged in user, or NULL for a no-authentication-required database.

When ATTACH-ing new database files to a connection, each newly attached database that is an authentication-required database is checked using the same username and password as provided to the main database. If that check fails, then the ATTACH command fails with an SQLITE_AUTH error.

Bootstrap

No-authentication-required database becomes an authentication-required database when the first user was added into the BDB database. This is called user authentication bootstrap. In bootstrap, the isAdmin parameter of the sqlite3_user_add() call must be true. After bootstrap, the first added user is logged into the database connection.

Transaction

BDB user authentication APIs sqlite3_user_add()/sqlite3_user_change()/sqlite3_user_delete() work in their own transaction. It results in an error to call these APIs inside a transaction.

Security Considerations

A BDB database is not considered as secure if it has only BDB user authentication applied status. The security issues are as follows:

- Anyone with access to the device can just open the database file in binary editor to see and modify the data.
- An authentication-required BDB database requires no authentication if opened by a version of BDB that omits the user authentication compile-time option

Due to the above issues BDB encryption has to be turned on when BDB user authentication is used. This requires the user to provide an encryption key before calling any of the authentication functions. If the database is encrypted, sqlite3_key_v2() must be called

first, with the correct decryption key, prior to invoking sqlite3_user_authenticate()/ sqlite3_user_add().

To open an existing, encrypted, authentication-required database, the call sequence is:

```
sqlite3_open_v2();
sqlite3_key_v2();
sqlite3_user_authenticate();
/* Database is now usable */
```

To create a new, encrypted, authentication-required database, the call sequence is:

```
sqlite3_open_v2();
sqlite3_key_v2();
sqlite3_user_add();
```

BDB SQL Key-store Based User Authentication

BDB SQL provides the key-store based user authentication to allow the user to work easily with encryption and user authentication together. In the key-store based user authentication, encryption becomes mandatory if user authentication is enabled, and user could just work with user authentication API only and without the knowledge of the encryption key.

You do this by storing the encryption key into a key-store file. The encryption key stored in the key-store file is encrypted. The key-store file, name ending with ".ks", is put under the same directory as the database environment. Each authenticated user has one entry in this key-store file. The entry contains the user's name and the encryption key. When sqlite3_user_authenticate() is called, if the encryption key is not applied to the database connection yet, BDB SQL will find the user's entry in the key-store file, compute the database encryption key with the user's password, then apply the encryption key to the database connection.

You can enable the key-store based user authentication by adding -DBDBSQL_USER_AUTHENTICATION_KEYSTORE compile option.

See the following sections for more information:

- Interface (page 7)
- Bootstrap (page 8)
- User Log In (page 8)
- Transaction (page 9)
- The Lock File (page 9)

Interface

The key-store user authentication APIs work the same with the non-keystore user authentication.

Bootstrap

No-authentication-required database becomes an authentication-required database until the first user is added into the BDB database. This is called user authentication bootstrap. In bootstrap, the isAdmin parameter of the sqlite3_user_add() call must be true. After bootstrap, the first added user is logged into the database connection. There are several cases related the the encryption key when doing key-store based user authentication bootstrap:

- If the database file does not exist yet and user does not provide his encryption key, a random generated key will be applied to the database and be stored into the key-store file. The call sequence is:

```
sqlite3_open_v2();
sqlite3_user_add();
```

> **Note**
>
> It is recommended for the user to backup the key-store file, especially when a generated random key is used.

- If the database file does not exist yet and user provides the encryption key, the encryption key provided will be applied to the database and be stored into the key-store file. The call sequence is:

```
sqlite3_open_v2();
sqlite3_key_v2();
sqlite3_user_add();
```

- If the database file already exists, the database is encrypted and the user provided the correct encryption key, the encryption key provided will be stored into the key-store file. The call sequence is:

```
sqlite3_open_v2();
sqlite3_key_v2();
sqlite3_user_add();
```

The bootstrap fails in case:

- The database file already exists, the database is encrypted and the user provided an incorrect encryption key.
- The database file already exists but the database is not encrypted.

User Log In

As encryption key is already in the key-store file, the user only needs to provide the user/password details to work with the encrypted database. BDB SQL will compute the encryption key from the key-store file and apply it to the database connection. The call sequence is as below:

```
sqlite3_open_v2();
sqlite3_user_authenticate();
/* Database is now usable */
```

The user can also provide the encryption key before the sqlite3_user_authenticate() call. In this case, BDB SQL will not visit the key-store file for the encryption key. The call sequence is as below:

```
sqlite3_open_v2();
sqlite3_key_v2();
sqlite3_user_authenticate();
/* Database is now usable */
```

Transaction

BDB user authentication API sqlite3_user_add()/sqlite3_user_change()/sqlite3_user_delete() works in their own transaction. It will result in an error if you call these APIs inside a transaction.

The Lock File

BDB key-store user authentication uses a locking file to ensure it behaves correctly in a multi-thread enviroment. In rare cases, if a system or application crash occurs while updating updating the key-store file, the locking file may not be cleaned and the next sqlite3_user_authenticate() call will be rejected. In this case, user needs to clean the .lck file under the database environment.

Unsupported PRAGMAs

The following PRAGMAs are not supported by the BDB SQL interface.

PRAGMA journal_mode
PRAGMA legacy_file_format

Also, PRAGMA fullfsync is always on for the BDB SQL interface. (This is an issue only for Mac OS X platforms.)

Changed PRAGMAs

The following PRAGMAs are available in the BDB SQL interface, but they behave differently in some way from standard SQLite.

PRAGMA auto_vacuum

The syntax for this PRAGMA is:

```
PRAGMA auto_vacuum
PRAGMA auto_vacuum = 0 | NONE | 1 | FULL | 2 | INCREMENTAL
```

Standard SQLite does not allow you to enable or disable auto-vacuum after a table has been created. Berkeley DB, however, allows you to change this at any time.

In the previous syntax, `0` and `NONE` both turn off auto vacuuming.

`1` or `FULL` causes full vacuuming to occur. That is, the BDB SQL interface will vacuum the entire database at each commit using a very low fill percentage (1%) in order to return

emptied pages to the file system. Because Berkeley DB allows you to call this PRAGMA at any time, it is recommended that you do not turn on FULL vacuuming because doing so can result in a great deal of overhead to your transaction commits.

If 2 or `INCREMENTAL` is used, then incremental vacuuming is enabled. The amount of vacuuming that is performed for incremental vacuum is controlled using the following PRAGMAs:

PRAGMA bdbsql_vacuum_fillpercent (page 12)
PRAGMA bdbsql_vacuum_pages (page 12)

Note that you can call PRAGMA incremental_vacuum (page 10) to perform an incremental vacuum operation on demand.

When performing vacuum operations, Berkeley DB defragments and repacks individual database pages, while SQLite only truncates the freelist pages from the database file.

For more information on auto vacuum, see PRAGMA auto_vacuum in the SQLite documentation.

PRAGMA cache_size

BDB SQL has a different default cache size than SQLite. There is a default value increase of PRAGMA cache size for SQL database from 2000 pages to 5000 pages.

PRAGMA incremental_vacuum

Performs incremental vacuum operations on demand. You can cause incremental vacuum operations to be performed automatically using PRAGMA auto_vacuum (page 9).

Note that for SQLite, this PRAGMA is used to specify the maximum number of pages to be freed during vacuuming. For Berkeley DB, you use `PRAGMA bdbsql_vacuum_pages` instead.

PRAGMA journal_size_limit

For standard SQLite, this pragma identifies the maximum size that the journal file is allowed to be.

Berkeley DB uses multiple journal files, Berkeley DB journal files are different to a SQLite journal file in that they contain information about multiple transactions, rather than a single transaction (similar to the SQLite WAL journal file). Over the course of the database's lifetime, Berkeley DB will probably create multiple journal files. A new journal file is created when the current journal file has reached the maximum size configured using the journal_size_limit pragma.

Note that a BDB SQL interface journal file is referred to as a log file in the Berkeley DB documentation.

Added PRAGMAs

The following PRAGMAs are added in the Berkeley DB SQL interface.

PRAGMA bdbsql_error_file

```
PRAGMA bdbsql_error_file [filename]
```

Redirects internal Berkeley DB error messages to the named file. If a relative path is specified to `[filename]`, then the path is interpreted as being relative to the current working directory.

If this PRAGMA is issued with no filename, then the current target for Berkeley DB error output is returned. By default, error messages are sent to `STDERR`.

This PRAGMA can be issued at any time; initial database access does not have to occur before this PRAGMA can be used.

PRAGMA bdbsql_lock_tablesize

```
PRAGMA bdbsql_lock_tablesize [= N]
```

Sets or reports the number of buckets in the Berkeley DB environment's lock object hash table.

This pragma must be called prior to opening/creating the database environment.

For more details, see get_lk_tablesize and set_lk_tablesize.

PRAGMA bdbsql_shared_resources

```
PRAGMA bdbsql_shared_resources [= N]
```

Sets or reports the maximum amount of memory (bytes) to be used by shared structures in the main environment region.

This pragma must be called prior to opening/creating the database environment.

For more details, see get_memory_max and set_memory_max.

PRAGMA bdbsql_single_process

```
PRAGMA bdbsql_single_process = boolean
```

To create a private environment rather than a shared environment, enable this pragma. The cache and other region files will be created in memory rather than using file backed shared memory.

In Berkeley DB SQL the default behavior is to allow a database to be opened and operated on by multiple processes simultaneously. When this pragma is enabled, accessing the same database from multiple processes simultaneously can lead to data corruption. Either option supports accessing a database using a single process multi-threaded application.

By default omit sharing is disabled. This pragma must be called prior to opening/creating the database environment. Because the setting is not persistent, you may need to invoke it before every database open, or define compile option `BDBSQL_OMIT_SHARING` instead.

For more information, see Shared memory region. Note that this pragma causes the DB_PRIVATE flag to be specified in the DB_ENV->open() method.

PRAGMA bdbsql_system_memory

```
PRAGMA bdbsql_system_memory [base segment ID]
```

Queries or sets a flag that causes the database's shared resources to be created in system shared memory. By default the database's shared resources are created in file-backed shared memory.

If a [base segment ID] is specified, the shared resources will be created using X/Open style shared memory interfaces. The [base segment ID] will be used as the starting ID for shared resources used by the database. Use different [base segment ID] values for different databases. It is possible for multi-process applications to use a single database by specifying the same [base segment ID] to this PRAGMA. Each connection needs to set this PRAGMA.

This PRAGMA may be used to set a [base segment ID] only before the first table is created in the database.

PRAGMA bdbsql_vacuum_fillpercent

```
PRAGMA bdbsql_vacuum_fillpercent [= N]
```

Sets or reports the page full threshold. Any page in the database that is at or below this percentage full is considered for vacuuming when `PRAGMA incremental_vacuum` is enabled. The value is specified as a percentage between 1 and 100. By default, pages 85% full and below are considered for vacuuming.

PRAGMA bdbsql_vacuum_pages

```
PRAGMA bdbsql_vacuum_pages [= N]
```

Sets or reports the maximum number of pages to be returned to the file system from the free page list when incremental vacuuming is enabled. By default, up to 128 pages are removed from the free list.

Page vacuuming is controlled using PRAGMA auto_vacuum (page 9).

PRAGMA large_record_opt

```
PRAGMA large_record_opt [= number bytes]
```

Enables optimized storage of large records. Any record larger than the given number of bytes will be stored in a new format that improves read and write performance for large records.

This pragma must be called before any data is added to the database, otherwise it will be ignored.

This optimization is incompatible with encryption, and `SQLITE_ReadUncommitted`.

This optimization stores large records in a folder called `__db_bl` in the journal folder. So if the database is manually moved to a new location, the folder with the large records most also be moved. The Online Backup Function will automatically backup up these records with the rest of the database.

PRAGMA multiversion

```
PRAGMA multiversion
```

Controls whether Multiversion Concurrency Control (MVCC) is on or off. You can not use this PRAGMA at any time during your application's runtime after your database tables have been accessed.

For more information on MVCC and snapshot isolation, see Using Multiversion Concurrency Control (page 25)

PRAGMA snapshot_isolation

```
PRAGMA snapshot_isolation
```

Controls whether snapshot isolation is turned on. This PRAGMA can be used at any time during your application's runtime *after* Multiversion Concurrency Control (MVCC) has been turned on.

For more information on MVCC and snapshot isolation, see Using Multiversion Concurrency Control (page 25)

PRAGMA statistics

```
PRAGMA statistics [= LOCK|LOG|MEM|MUTEX|REP]
```

Prints various statistics to the entered subsystem, or general environment statistics if no subsystem is entered. The subsystems are locking, logging, memory pool, mutex and replication. The output is printed to stdout, unless the statistics_file PRAGMA is used to set a file to which the output is directed.

PRAGMA statistics_file

```
PRAGMA statistics_file [= filename]
```

Sets a file to which the output from the statistics PRAGMA is printed. Otherwise the output is printed to stdout. The file will be automatically created if it does not exist, and output will be appended to the end of the file if it does exist.

PRAGMA trickle

```
PRAGMA trickle [percent]
```

Ensures that at least the specified percentage of pages in the shared cache are clean. This can cause pages that have been modified to be flushed to disk.

The trickle functionality enables an application to ensure that a page is available for reading new information into the shared cache without waiting for a write operation to complete.

PRAGMA txn_bulk

```
PRAGMA TXN_BULK
```

Enables transactional bulk loading optimization. For more information, see Using Bulk Loading (page 25).

Replication PRAGMAs

Fourteen PRAGMAs were added to manage and control replication. They are described in Using Replication with the SQL API (page 31):

Note

If you are using these replication PRAGMAs and you want to perform a backup, there is an additional backup step for the pragma file. See Backing Up Berkeley DB SQL Databases (page 42) for more information.

- PRAGMA replication (page 33)
- PRAGMA replication_ack_policy (page 34)
- PRAGMA replication_ack_timeout (page 35)
- PRAGMA replication_get_master (page 35)
- PRAGMA replication_initial_master (page 35)
- PRAGMA replication_local_site (page 35)
- PRAGMA replication_num_sites (page 36)
- PRAGMA replication_perm_failed (page 36)
- PRAGMA replication_priority (page 36)
- PRAGMA replication_remote_site (page 36)
- PRAGMA replication_remove_site (page 37)
- PRAGMA replication_site_status (page 37)
- PRAGMA replication_verbose_output (page 37)
- PRAGMA replication_verbose_file (page 37)

PRAGMA bdbsql_userauth_add

```
PRAGMA bdbsql_userauth_add = "user:password:isAdmin";
```

Creates a new user. isAdmin should only be 0 (non-admin user) or 1 (admin user).

Note

':' is not alowed to appear in the user name/password when working with BDB SQL user authentication pragmas.

PRAGMA bdbsql_user_login

```
PRAGMA bdbsql_user_login = "user:password";
```

Authenticates a user.

Note

':' is not alowed to appear in the user name/password when working with BDB SQL user authentication pragmas.

PRAGMA bdbsql_user_edit

```
PRAGMA bdbsql_user_edit = "user:password:isAdmin";
```

Change a user's login credentials or admin privilege. isAdmin should only be 0 (non-admin user) or 1 (admin user).

Note

':' is not alowed to appear in the user name/password when working with BDB SQL user authentication pragmas.

PRAGMA bdbsql_user_delete

```
PRAGMA bdbsql_user_delete = "user";
```

Deletes a user.

Note

':' is not alowed to appear in the user name/password when working with BDB SQL user authentication pragmas.

Miscellaneous Differences

The following miscellaneous differences also exist between the BDB SQL interface and SQLite:

- The BDB SQL interface does not support the `IMMEDIATE` keyword (`BEGIN IMMEDIATE` behaves just like `BEGIN`).
- When an exclusive transaction is active, it will block any new transactions from beginning (they will be blocked during their first operation until the exclusive transactions commits or aborts). Non-exclusive transactions that are active when the exclusive transaction begins will not be able to execute any more operations without being blocked until the exclusive transactions finishes.
- Enabling MVCC mostly disables exclusive transactions. Exclusive transactions can still be used, but they will run concurrently with regular transactions, even ones that write to the database. The only advantage of exclusive transactions in this case is that two exclusive transactions will be forced to run in serial, and that if an exclusive transaction and non-exclusive transaction experience deadlock, then the non-exclusive transaction will always be the transaction forced to release its locks.

  For more information on MVCC and snapshot isolation, see Using Multiversion Concurrency Control (page 25)
- There are differences in how the two products work in a concurrent application that will cause the BDB SQL interface to deadlock where SQLite would result in a different error. This

is because the products use different locking paradigms. See Locking Notes (page 21) for more information.

- The BDB SQL does not call the busy callback when a session attempts to operate the same database page that another session has locked. It blocks instead. This means that the functions `sqlite3_busy_handler` and `sqlite3_busy_timeout` are not effective in BDB SQL.
- The BDB SQL does not support two phase commit across databases. Attaching to multiple databases can lead to inconsistency after recovery and undetected deadlocks when accessing multiple databases from concurrent transactions in different order. Hence, applications must ensure that they access databases in the same order in any transaction that spans multiple databases. Else, a deadlock can occur that causes threads to block, and the deadlock will not be detected by Berkeley DB.
- In BDB SQL, when two sessions accessing the same database perform conflicting operations on the same page, one session will be blocked until the conflicting operations are resolved. For example,

  Session 1:

  ```
  dbsql> insert into a values (4);
  dbsql> begin;
  dbsql> insert into a values (5);
  ```

  Session 2:

  ```
  dbsql> select * from a;
  ```

  What happens here is that Session 2 is blocked until Session 1 commits the transaction.

  Session 1:

  ```
  dbsql> commit;
  ```

  Session 2:

  ```
  dbsql> select * from a;
  4
  5
  ```

  Under such situations in SQLite, operations poll instead of blocking, and a callback is used to determine whether to continue polling.
- By default, you always only have a single database file when you use BDB SQL interface SQL, just as you do when you use SQLite. However, you can configure BDB SQL interface at compile time to create one BDB SQL interface database file for each SQL table that you create. How to perform this configuration is described in the *Berkeley DB Installation and Build Guide*.
- BDB SQL has a different default cache size than SQLite. PRAGMA cache size for SQL database has a default value increase. The default PRAGMA cache size for SQL databases is increased from 2000 pages to 5000 pages.

Berkeley DB Concepts

If you are a SQLite user who is migrating to the BDB SQL interface, then there are a few Berkeley DB-specific concepts you might want to know about.

- Environments. The directory that is created alongside your database file, and which ends with the "-journal" suffix, is actually a Berkeley DB environment directory.
- The Locking Subsystem

  The Berkeley DB library implements locking in a different way to SQLite. SQLite implements locking at a database level - any operation will take a lock on the entire database. Berkeley DB implements a scheme called page level locking. The database divides data into relatively small blocks. Each block corresponds to a page in database terms. Each block can contain multiple pieces of user information. Berkeley DB takes locks on individual pages. This allows for greater concurrency in applications, but means that applications are more likely to encounter deadlocks.

  See: Locking Notes (page 21) for more information.
- The Journal Subsystem

  The BDB SQL interface implements write ahead logging (WAL), it stores journal files differently to the SQLite WAL implementation. BDB SQL interface rolls over journal files when they get to a certain size (default 10MB). It is possible for the to be multiple journal files active at one time with BDB SQL interface.

Encryption

When encryption is enabled, the Berkeley DB SQL API uses native Berkeley DB encryption to assure the security of your data. As usual, the Berkeley DB SQL API is almost identical to the SQLite API, so you can use the syntax of the SQLite Encryption Extension to interact with your encrypted data.

Berkeley DB encryption

Berkeley DB supports encryption using the Rijndael/AES algorithm. It 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. For more information on BDB encryption, see the Berkeley DB Programmer's Reference Guide.

SQLite Encryption Extension

To learn about the APIs which are used in the SQLite Encryption Extension (SEE), see the official SQLite Documentation Page.

Note

The Berkeley DB SQL interface does not support the sqlite3_rekey method.

Using Sequences

You can use sequences with the SQL API. Sequences provide for an arbitrary number of increasing or decreasing integers that persist across database accesses. Use sequences if you need to create unique values in a highly efficient and persistent way.

To create and access a sequence, you must use SQL functionality that is unique to the BDB SQL interface; no corresponding functionality exists in SQLite. The sequence functionality is implemented using SQLite function plugins, as such it is necessary to use the 'select' keyword as a prefix to all sequence APIs.

The SQL API sequence support is a partial implementation of the sequence API defined in the SQL 2003 specification.

The following sections describe the BDB SQL interface sequence API.

create_sequence

Creates a new sequence. A name is required, all other parameters are optional. For example:

```
SELECT create_sequence("my_sequence", "start", 100, "incr", 10,
                       "maxvalue", 300);
```

This creates a sequence called `my_sequence` starting at 100 and incrementing by 10 until it reaches 300.

```
SELECT create_sequence("my_decr_sequence", "incr", -100,
                       "minvalue", -10000);
```

This creates a sequence call `my_decr_sequence` starting at 0 and decreasing by 100 until it reaches -10000.

Parameters are:

- `name`

  Required parameter that provides the name of the sequence. It is an error to create a sequence with another name that is currently in use within the database.

- `start`

  The starting value for the sequence. If this parameter is not provided it is set to `minvalue` if an incrementing sequence is used, and `maxvalue` if a decrementing sequence is used. If neither of those parameters are set then `0` is used.

- `minvalue`

  The lowest value generated by the sequence. If this parameter is not provided and a decrementing sequence is created, then INT64_MIN is used.

- `maxvalue`

The largest value generated by the sequence. If this parameter is not provided and an incrementing sequence is created, then INT64_MAX is used.

- `incr`

  The amount the sequence is incremented for each get operation. This value can be positive or negative. If this parameter is not provided, then 1 is used.

- `cache`

  Causes each handle to keep a cache of sequence values. So long as there are values available in the cache, retrieving the next value is cheap and does not lead to contention between handles.

  Sequences with caches cannot be created or dropped within an explicit transaction.

  Operations on caching sequences are not transactionally protected. That is, a rollback will not result in a value being returned to the sequence.

  Sequences with caches do not support the `currval` function.

  The parameter following the cache parameter must be an integer value specifying the size of the cache.

nextval

Retrieves the next value from the named sequence. For example:

```
SELECT nextval("my_sequence");
```

currval

Retrieves the last value that was returned from the named sequence. For example:

```
SELECT currval("my_sequence");
```

drop_sequence

Removes the sequence. For example:

```
SELECT drop_sequence("my_sequence");
```

Differences for Users of other SQL Engines

If you are used to a SQL implementation from other SQL engine (such as Oracle's RDBMS), the SQL used by the BDB SQL interface (which is the same as used by SQLite) may hold some surprises for you.

Some things in particular to take note of:

- Datatyping is weaker in SQLite than it is with standard SQL. For example, SQLite does not enforce the length of a `VARCHAR`. While standard SQL will truncate a `VARCHAR` that is too

long, you could (for example) declare a `VARCHAR(10)` then put 500 characters in it without any truncation, ever.

SQLite datatyping is described in detail on the Datatypes in SQLite Version 3 page.

- Do not use autocommit with SQLite. Instead, use `begin exclusive` and then `commit`.
- How NULLs are handled in SQLite may be different from what you are used to. See NULL Handling in SQLite Versus Other Database Engines for details.
- There are some features of SQL that SQLite does not support. For more information, see SQL Features That SQLite Does Not Implement.

Chapter 2. Locking Notes

There are some important performance differences between the BDB SQL interface and SQLite, especially in a concurrent environment. This chapter gives you enough information about how the BDB SQL interface uses its database, as opposed to how SQLite uses its database, in order for you to understand the difference between the two interfaces. It then gives you some advice on how to best approach working with the BDB SQL interface in a multi-threaded environment.

If you are an existing user of SQLite, and you care about improving your application performance when using the BDB SQL interface in a concurrent situation, you should read this chapter. Existing users of Berkeley DB may also find some interesting information in this chapter, although it is mostly geared towards SQLite users.

Internal Database Usage

The BDB SQL interface and SQLite do different things when it comes to locking data in their databases. In order to provide ACID transactions, both products must prevent concurrent access during write operations. Further, both products prevent concurrent access by obtaining software level locks that allow only the current holder of the lock to perform write access to the locked data.

The difference between the two is that when SQLite requires a lock (such as when a transaction is underway), it locks the entire database and all tables. (This is known as *database level locking*.) The BDB SQL interface, on the other hand, only locks the portion of the table being operated on within the current transactional context (this is known as *page level locking*). In most situations, this allows applications using the BDB SQL interface to operate concurrently and so have better read/write throughput than applications using SQLite. This is because there is less lock contention.

By default, one Berkeley DB logical database is created within the single database file for every SQL table that you create. Within each such logical database, each table row is represented as a Berkeley DB key/data pair.

This is important because the BDB SQL interface uses Berkeley DB's Transaction Data Store product. This means that Berkeley DB does not have to lock an entire database (all the tables within a database file) when it acquires a lock. Instead, it locks a single Berkeley DB database page (which usually contains a small sub-set of rows within a single table).

The size of database pages will differ from platform to platform (you can also manually configure this), but usually a database page can hold multiple key/data pairs; that is, multiple rows from a SQL table. Exactly how many table rows fit on a database page depends on the size of your page and the size of your table rows.

If you have an exceptionally small table, it is possible for the entire table to fit on a single database page. In this case, Berkeley DB is in essence forced to serialize access to the entire table when it requires a lock for it.

Note, however, that the case of a single table fitting on a single database page is very rare, and it in fact represents the abnormal case. Normally tables span multiple pages and so

Berkeley DB will lock only portions of your tables. This locking behavior is automatic and transparent to your application.

Lock Handling

There is a difference in how applications written for the BDB SQL interface handle deadlocks as opposed to how deadlocks are handled for SQLite applications. For the SQLite developer, the following information is a necessary review in order to understand how the BDB SQL interface behaves differently.

From a usage point of view, the BDB SQL interface behaves in the same way as SQLite in shared cache mode. The implications of this are explained below.

SQLite Lock Usage

As mentioned previously in this chapter, SQLite locks the entire database while performing a transaction. It also has a locking model that is different from the BDB SQL interface, one that supports multiple readers, but only a single writer. In SQLite, transactions can start as follows:

- `BEGIN`

  Begins the transaction, locking the entire database for reading. Use this if you only want to read from the database.

- `BEGIN IMMEDIATE`

  Begins the transaction, acquiring a "modify" lock. This is also known as a RESERVED lock. Use this if you are modifying the database (that is, performing `INSERT`, `UPDATE`, or `DELETE`). RESERVED locks and read locks can co-exist.

- `BEGIN EXCLUSIVE`

  Begins the transaction, acquiring a write lock. Transactions begun this way will be written to the disk upon commit. No other lock can co-exist with an exclusive lock.

The last two statements are a kind of a contract. If you can get them to complete (that is, not return `SQLITE_LOCKED`), then you can start modifying the database (that is, change data in the in-memory cache), and you will eventually be able to commit (write) your modifications to the database.

In order to avoid deadlocks in SQLite, programmers who want to modify a SQLite database start the transaction with `BEGIN IMMEDIATE`. If the transaction cannot acquire the necessary locks, it will fail, returning `SQLITE_BUSY`. At that point, the transaction falls back to an unlocked state whereby it holds no locks against the database. This means that any existing transactions in a RESERVED state can safely wait for the necessary EXCLUSIVE lock in order to finally write their modifications from the in-memory cache to the on-disk database.

The important point here is that so long as the programmer uses these locks correctly, he can assume that he can proceed with his work without encountering a deadlock. (Assuming that all database readers and writers are also using these locks correctly.)

Lock Usage with the BDB SQL Interface

When you use the BDB SQL interface, you can begin your transaction with `BEGIN` or `BEGIN EXCLUSIVE`.

Note that the `IMMEDIATE` keyword is ignored in the BDB SQL interface (`BEGIN IMMEDIATE` behaves like `BEGIN`).

When you begin your transaction with `BEGIN`, Berkeley DB decides what kind of a lock you need based on what you are doing to the database. If you perform an action that is read-only, it acquires a read lock. If you perform a write action, it acquires a write lock.

Also, the BDB SQL interface supports multiple readers *and* multiple writers. This means that multiple transactions can acquire locks as long as they are not trying to modify the same page. For example:

Session 1:

```
dbsql> create table a(x int);
dbsql> begin;
dbsql> insert into a values (1);
dbsql> commit;
```

Session 2:

```
dbsql> create table b(x int);
dbsql> begin;
dbsql> insert into b values (1);
dbsql> commit;
```

Because these two sessions are operating on different pages in the Berkeley DB cache, this example will work. If you tried this with SQLite, you could not start the second transaction until the first had completed.

However, if you do this using the BDB SQL interface:

Session 1:

```
dbsql> begin;
dbsql> insert into a values (2);
```

Session 2:

```
dbsql> begin;
dbsql> insert into a values (2);
```

The second session blocks until the first session commits the transaction. Again, this is because both sessions are operating on the same database page(s). However, if you simultaneously attempt to write pages in reverse order, you can deadlock. For example:

Session 1:

```
dbsql> begin;
```

```
dbsql> insert into a values (3);
dbsql> insert into b values (3);
```

Session 2:

```
dbsql> begin;
dbsql> insert into b values (3);
dbsql> insert into a values (3);
Error: database table is locked
```

What happens here is that Session 1 is blocked waiting for a lock on table b, while Session 2 is blocked waiting for a lock on table a. The application can make no forward progress, and so it is deadlocked.

When such a deadlock is detected one session loses the lock it got when executing its last statement, and that statement is automatically rolled back. The rest of the statements in the session will still be valid, and you can continue to execute statements in that session. The session that does not lose its lock to deadlock detection will continue to execute as if nothing happened.

Assume Session 2 was sacrificed to deadlock detection, no value would be inserted into a and an error will be returned. But the insertion of value 3 into b would still be valid. Session 1 would continue to wait while inserting into table b until Session 2 either commits or aborts, thus freeing the lock it has on table b.

When you begin your transaction with `BEGIN EXCLUSIVE`, the session is never aborted due to deadlock or lock contention with another transaction. Non-exclusive transactions are allowed to execute concurrently with the exclusive transaction, but the non-exclusive transactions will have their locks released if deadlock with the exclusive transaction occurs. If two or more exclusive transactions are running at the same time, they will be forced to execute in serial.

If Session 1 was using an exclusive transaction, then Session 2 would lose its locks when deadlock is detected between the two. If both Session 1 and Session 2 start an exclusive transaction, then the last one to start the exclusive transaction would be blocked after executing `BEGIN EXCLUSIVE` until the first one is committed or aborted.

Chapter 3. Berkeley DB Features

When using the Berkeley DB SQL API, there are features that you can manage using PRAGMAs that are unique to Berkeley DB. This chapter discusses these features.

Using Bulk Loading

Bulk loading is an I/O optimization feature that is useful when loading large amounts of data to the database inside of a single transaction. The bulk load optimization avoids writing some log records to the journal file. This can provide significant performance benefits when loading a large number of new rows into a database.

When you use bulk load, nested transactions are disabled. This means that you cannot undo any single operation. Instead, if you want to undo a given operation, you must undo everything performed within the transaction.

It is not possible to use the bulk load functionality when replication is enabled.

To enable bulk loading, use:

PRAGMA TXN_BULK = 0 | 1;

Default value is `0`, which means the PRAGMA is turned off and so bulk loading is not in use. `1` means bulk loading is in use.

Using Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) enables snapshot isolation. Snapshot isolation means that whenever a transaction would take a read lock on a page, it makes a copy of the page instead, and then performs its operations on that copied page. This frees other writers from blocking due to a read locks held by other transactions.

You should use snapshot isolation whenever you have a lot of read-only transactions operating at the same time that read-write transactions. In this case, snapshot isolation will improve transaction throughput, albeit at the cost of greater resource usage.

MVCC is described in more detail in Snapshot Isolation.

To use MVCC, you must enable it before you access any database tables. Once MVCC is enabled, you can turn snapshot isolation on and off at anytime during the life of your application.

To turn MVCC on, use:

PRAGMA multiversion = on | off;

This PRAGMA must be enabled before you access any database tables during your application runtime, or an error is returned. Turning MVCC on automatically enables snapshot isolation. By default MVCC is turned off.

Once MVCC is enabled, you can turn snapshot isolation on and off using:

PRAGMA snapshot_isolation = on | off;

This PRAGMA can be used at any time during the life of your application *after* MVCC has been turned on. If you attempt to enable or disable snapshot isolation before MVCC is enabled, an error is returned.

Selecting the Page Size

When using the BDB SQL interface, you configure your database page size in exactly the same way as you do when using SQLite. That is, use `PRAGMA page_size` to report and set the page size. This PRAGMA must be called before you create your first SQLite table. See the PRAGMA page_size documentation for more information.

When you use `PRAGMA cache_size` to size your in-memory cache, you provide the cache size in terms of a number of pages. Therefore, your database page size influences how large your cache is, and so determines how much of your database will fit into memory.

The size of your pages can also affect how efficient your application is at performing disk I/O. It will also determine just how fine-grained the fine-grained locking actually is. This is because Berkeley DB locks database pages when it acquires a lock.

Note that the default value for your page size is probably correct for the physical hardware that you are using. In almost all situations, the default page size value will give your application the best possible I/O performance. For this reason, tuning the page size should rarely, if ever, be attempted.

That said, when using the BDB SQL interface, the page size affects how much of your tables are locked when read and/or write locks are acquired. (See Internal Database Usage (page 21) for more information.) Increasing your page size will typically improve the bandwidth you get accessing the disk, but it also may increase contention if too many key data pairs are on the same page. Decreasing your page size frequently improves concurrency, but may increase the number of locks you need to acquire and may decrease your disk bandwidth.

When changing your page size, make sure the value you select is a power of 2 that is greater than 512 and less than or equal to 64KB. (Note that the standard SQLite `MAX_PAGE_SIZE` limit is not examined for this upper bound.)

Beyond that, there are some additional things that you need to consider when selecting your page size. For a thorough treatment of selecting your page size, see the section on Selecting a page size in the *Berkeley DB Programmer's Reference Guide*.

Controlling the Number of Accumulated Log Files

In Berkeley DB SQL interface, the `pragma wal_autocheckpoint` can be used to control the total number of log files that get accumulated before an automatic purge happens. For example, if we set `pragma wal_checkpoint=N`, and when the current amount of data written to logs since the last checkpoint equals or exceeds "N" pages, a checkpoint happens. If the log file auto-removing is enabled (which is enabled by default), the accumulated log files are auto-purged.

In Berkeley DB SQL interface, the default value of `wal_autocheckpoint` is 512 pages. With the default page size as 4096 bytes, it means when the current amount of data written to logs since the last checkpoint equals or exceeds 2 MB, a checkpoint happens and the accumulated log files purge. As in Berkeley DB SQL interface, by default a single log file has the maximium size as 2 MB, so we will at most have two 2 MB log files in a Bekeley DB environment.

You can make changes to the default values. For example, if you change `wal_autocheckpoint` to 256 pages with the page size as 4096 bytes, it means when the current amount of data written to logs since the last checkpoint equals or exceeds 1 MB, a checkpoint happens and the accumulated log files get purged. You can also change the single log file size to 256 KB (check the usage of `PRAGMA journal_size_limit`). With this setting, you will have maximum five 256 KB log files in the Berkeley DB environment.

Note

- If there are huge transactions, the Berkeley DB accumulated log files will be as many as needed.
- User can turn off the auto-checkpointing (for example, by `PRAGMA wal_autocheckpoint = -1`). In this case, users must have their own management on the accumulated log files.
- `Pragma wal_autocheckpoint` is not persistent and will reset to default after we reset the database connection.

Chapter 4. Using DB_CONFIG to configure the Berkeley DB SQL interface

In almost all cases, there is no need for you to directly configure Berkeley DB resources; instead, you can use the same configuration techniques that you always use for SQLite, or use the Berkeley DB PRAGMAs describe Berkeley DB SQL: The Absolute Basics (page 1), and Replication PRAGMAs (page 33). The Berkeley DB SQL interface will take care of the rest.

However, there are a few configuration activities that some unusually large or busy installations might need to make and for which there is no SQLite equivalent. This chapter describes those activities.

Introduction to Environments

Before continuing with this section, it is necessary for you to have a high-level understanding of Berkeley DB's environments.

In order to manage its resources (data, shared cache, locks, and transaction logs), Berkeley DB often uses a directory that is called the *Berkeley DB environment*. As used with the BDB SQL interface, environments contain log files and the information required to implement a shared cache and fine-grained locking. This environment is placed in a directory that appears on the surface to be a SQLite rollback file.

That is, if you use BDB SQL interface to create a database called `mydb.db`, then a directory is created alongside of it called `mydb.db-journal`. Normally, SQLite creates a journal file only when a transaction is underway, and deletes this file once the transaction is committed or rolled back. However, that is not what is happening here. The BDB SQL interface journal directory contains important Berkeley DB environment information that is meant to persist between transactions and even between process runtimes. So it is very important that you do *not* delete the contents of your Berkeley DB journal directory. Doing so will cause improper operation and could lead to data loss.

Note that the environment directory is also where you put your `DB_CONFIG` file. This file can be used to configure additional tuning parameters of Berkeley DB, if its default behavior is not appropriate for your application. For more information on the `DB_CONFIG` file, see the next section.

Note

Experienced users of Berkeley DB should be aware that neither `DB_USE_ENVIRON` nor `DB_USE_ENVIRON_ROOT` are specified to `DB_ENV->open()`. As a result, the `DB_HOME` environment variable is ignored. This means that the BDB SQL interface will always create a database in the location defined by the database name given to the BDB SQL interface.

The DB_CONFIG File

You can configure most aspects of your Berkeley DB environment by using the `DB_CONFIG` file. This file must be placed in your environment directory. When using the BDB SQL interface, this

is the directory created alongside of your database. It has the same name as your database, followed by a `-journal` extension. For example, if your database is named `mydb.db`, then your environment directory is created next to the `mydb.db` file, and it is called `mydb.db-journal`.

If a `DB_CONFIG` file exists in your environment directory, it will be read when your environment is opened. This happens when your application starts up and creates its first connection to the database.

Configurations set in the DB_CONFIG file usually appear as the name of the configuration's C function, as documented in Berkeley DB C API Reference Guide, followed by the values of the function arguments. For example, you can use the set_lg_dir DB_CONFIG parameter to configure the directory in which log files are stored like this:

```
set_lg_dir ../myLogDir
```

In some cases, you must either specify a configuration option before the environment is created, or the environment must be re-created before the configuration option will take effect. The documentation for each configuration option will indicate where this is true.

Creating the DB_CONFIG File Before Creating the Database

In order to provide the `DB_CONFIG` file before the database is first created, physically make the journal directory in the correct location in your filesystem (this is wherever you want to place your database file), and put the `DB_CONFIG` file there before you create your database. For example, if you plan to create a database named `mydb.db`, then in the same directory as the database, create a directory named `mydb.db-journal`.

Re-creating the Environment

Some `DB_CONFIG` parameters require you to re-create your environment before they take effect. The `DB_CONFIG` parameter descriptions indicates where this is the case.

To re-create your environment:

- Make sure the `DB_CONFIG` file contains the following line:

  ```
  add_data_dir ..
  ```

  (This line should already be in the `DB_CONFIG` file.)

- Run the db_recover command line utility. If you run it from within your environment (`-journal`) directory, no command line arguments are required. If you run it from outside your environment directory, use the `-h` parameter to identify the location of the environment directory:

  ```
  db_recover -h /some/path/to/mydb.db-journal
  ```

Configuring the In-Memory Cache

SQLite provides an in-memory cache which you size according to the maximum number of database pages that you want to hold in memory at any given time.

Berkeley DB also provides an in-memory cache that performs the same function as SQLite. You can configure this cache using the exact same PRAGMAs as you are used to using with SQLite. See PRAGMA cache_size and PRAGMA default_cache_size for details. As is the case with SQLite, you use these PRAGMAs to describe the total number of pages that you want in the cache.

However, you can also configure Berkeley DB's in-memory cache size using the `DB_CONFIG` file. When you do this, you do not specify the cache size in terms of the number of pages to be contained there. Instead, you configure it in terms of the total number of bytes that you want it to be able to hold.

Like the in-memory cache for SQLite, you can change Berkeley DB's cache size at any time. However, if you want to manage it using the `DB_CONFIG` file, then you must restart your application. In addition, you must re-create your environment in order for the new value to take effect. See The DB_CONFIG File (page 28) for details.

To configure your cache size using the `DB_CONFIG` file, use the set_cachesize parameter.

Chapter 5. Using Replication with the SQL API

The Berkeley DB SQL interface allows you to use Berkeley DB's replication feature. You configure and start replication using PRAGMAs that are specific to the task.

This chapter provides a high-level introduction of Berkeley DB replication. It then shows how to configure and use replication with the SQL API.

For a more detailed description of Berkeley DB replication, see:

- *Berkeley DB Getting Started with Replicated Applications*
- *Berkeley DB Programmer's Reference Guide*

Replication Overview

Berkeley DB's replication feature allows you to automatically distribute your database write operations to one or more read-only *replicas*. For this reason, BDB's replication implementation is said to be a *single master, multiple replica* replication strategy.

A single replication master and all of its replicas are referred to as a *replication group*. Each replication group can have one and only one master site.

When discussing Berkeley DB replication, we sometimes refer to *replication sites*. This is because most production applications place each of their replication participants on separate physical machines. In fact, each replication participant must be assigned a hostname/port pair that is unique within the replication group.

Note that under the hood, the unit of replication is the environment. That is, data is replicated from one Berkeley DB environment to one or more other Berkeley DB environments. However, when used with the BDB SQL interface, you can think of this as replicating between Berkeley DB databases, because the BDB SQL interface results in a single database file for each environment.

Replication Masters

Every replication group has one and only one master. The master site is where you perform write operations. These operations are then automatically replicated to the other sites in the replication group. Because the other replica sites in the replication group are read-only, it is an error for you to attempt to perform write operatons on them.

The replication master is usually automatically selected by the replication group using elections. Replication elections simply determine which replication site has the most up-to-date copy of the data, and so is in the best position to serve as the master site.

Note that when you initially start up your BDB SQL replicated application, you must explicitly designate a specific site as the master. Over time, the master site can move from one environment to the next. For example, if the master site is shut down, becomes unavailable, or a network partition causes it to lose contact with the rest of the replication group, then the

replication group will elect a new master if it can successfully hold an election. When the old master comes back online, it rejoins the replication group as a read-only replica site.

Also, if you are enabling replication for an existing database, then that database must be designated as the master. Doing this is required; otherwise the entire contents of the existing database might be deleted during the replication startup process.

Elections

A replication group selects the master site by holding an election. In simplistic terms, each participant in the replication group votes on who it believes has the most up-to-date version of the data that the replication group is managing. The site that receives the most number of votes becomes the master site, and all data write activity must occur there.

In order to hold an election, the replication group must have a quorum. In order to achieve a quorum, a simple majority of the sites must be available to select the master. That is, *n/2 + 1* sites must be available, where *n* is the total number of replication group participants. By requiring a simple majority, the replication group avoids the possibility of simultaneously running with two master sites due to a network partition.

If a replication group cannot select a master, then it can only be used in read-only mode.

Durability Guarantees

Durability is a term that means data modifications have met some pre-defined set of guarantees that the modifications will remain persistent across application run times. Usually, this means that there is some assurance that the data modification has been written to stable storage (that is, written to a hard drive).

For replicated BDB SQL applications, the durability guarantee is enhanced because data modifications are also replicated to those environments that are participating in the replication group. This ensures higher data durability than non-replicated applications by placing data in multiple environments that usually reside on separate physical machines.

Permanent Message Handling

Permanent messages are created by replication masters as a part of a transactional commit operation. When a replica receives a message that is marked as permanent, it knows that the message affects transactional integrity. Receipt of a permanent message means that the replica must send a message acknowledgment back to the master server because the master *might be* waiting for the acknowledgment before it considers the transaction commit to be complete.

Whether the master is actually waiting for message acknowledgement depends on the acknowledgement policy in effect for the replication group. Policies can range from `NONE` (the master will not wait for any acknowledgements before completing the transaction) to `ALL` (the master will wait for acknowledgements from all replicas before completing the transaction).

Acknowledgements are only sent back to the master once the replica has completed applying the message to its local environment. Therefore, the stronger your acknowledgement policy,

the stronger you durability guarantee. On the other hand, the stronger your acknowledgement policy, the slower your application's write throughput will be.

In addition to setting an acknowledgement policy, you can also set an acknowledgment timeout. This time limit is set in microseconds and it represents the length of time the master will wait to satisfy its acknowledgement policy for each transaction commit. If this timeout value is not met, the transaction is still committed locally to the master, but is not yet considered durable across the replication group. Your code should take whatever actions are appropriate for that transaction. If enough other sites are available to meet the acknowledgement policy, the transaction will become durable after more time has passed.

You set acknowledgement policies and acknowledgement timeouts using PRAGMAs. See PRAGMA replication_ack_policy (page 34) and PRAGMA replication_ack_timeout (page 35). In addition, you can examine how frequently your transactions do not achieve durability within the acknowledgement timeout by using PRAGMA replication_perm_failed (page 36).

Two-Site Replication Groups

In a replication group that consists of exactly two sites, both sites must be available in order to achieve a quorum. Without a quorum, a new master site cannot be elected. This means that if the master site is unable to participate in the replication group, then the remaining read-only replica cannot become the master site.

In other words, if you have a group that consists of exactly two sites, if you lose your master site then the replication group must exist in read-only mode until the master site becomes available again.

Replication PRAGMAs

To control replication when using the Berkeley DB SQL interface, you use the following PRAGMAs. For an example of how to use these, see Replication Usage Examples (page 38).

PRAGMA replication

```
PRAGMA replication=ON|OFF
```

Enables the local environment to participate in replication. You only need to specify this PRAGMA once to turn on replication; all future uses of the database will automatically enable replication.

Before invoking this PRAGMA for a brand new database (one that has never been opened), you must invoke the `replication_local_site` PRAGMA and then either the `replication_initial_master` or the `replication_remote_site` PRAGMA. These actions define the way this site fits into the replication group.

If you are enabling replication for an existing database, it must become the initial master for a new replication group. You must invoke the `replication_local_site` PRAGMA followed by the `replication_initial_master` PRAGMA before enabling replication.

If you use this PRAGMA to turn off replication, then replication is completely disabled for the environment. In order to enable replication again, you follow the procedure used to

enable replication on an existing database; that is, invoke the `replication_local_site` PRAGMA followed by the `replication_initial_master` PRAGMA, followed by `PRAGMA replication=ON`.

PRAGMA replication_ack_policy

```
PRAGMA replication_ack_policy=all|all_available|none|one|quorum
```

Sets the replication group's acknowledgement policy. If no policy is specified, then this PRAGMA returns the current acknowledgement policy. This PRAGMA may be called at any time during the life of the application.

Acknowledgement policies are used to describe how permanent messages will be handled. A message is considered to be permanent if the master site has received enough responses (acknowledgements) from its replicas. Once the master is satisfied that a message is permanent, it considers the transaction that generated the message to be complete. Therefore, the acknowledgement policy you set affects your application's write throughput, as well as the durability strength of your transactions.

Message acknowledgements must be returned within a set timeout period. See PRAGMA replication_ack_timeout (page 35) for information on how to manage this timeout. Also, PRAGMA replication_perm_failed (page 36) provides useful diagnostic information that you can use to fine-tune your acknowledgement policy.

See Permanent Message Handling (page 32) for more information on message acknowledgements.

Possible permanent acknowledgement policies are:

- `all`

  All sites belonging to the replication group must acknowledge the message before the generating transaction is considered to be complete. This provides the strongest possible durability guarantee, but also the slowest possible write throughput.

  Note that if any site is shutdown or otherwise cannot be reached due to networking errors, this acknowledgement policy will prevent your application from considering any transactions durable.

- `all_available`

  All currently connected sites must acknowledge the message.

- none

  The master will not wait for any message acknowledgements before completing the transaction. This results in the fastest possible write throughput, but also the weakest durability guarantee.

- one

  At least one site must acknowledge the message before the transaction is considered to be complete.

- `quorum`

  The master waits until it has received acknowledgements from the minimum number of electable sites sufficient to ensure the record change is durable in the event of an election. This is the default acknowledgement policy.

PRAGMA replication_ack_timeout

```
PRAGMA replication_ack_timeout=<timeout in microseconds>
```

Sets the replication site's acknowledgement timeout value. If the acknowledgement policy cannot be met within this timeout period, then the encompassing transaction is not considered to be complete. This PRAGMA may be called at any time during the life of the application. The default is 1 second (1000000).

If no timeout is provided, then this PRAGMA returns the replication site's current message acknowledgement timeout value.

For information on specifying acknowledgement policies, see PRAGMA replication_ack_policy (page 34). Also, PRAGMA replication_perm_failed (page 36) provides useful diagnostic information that you can use to fine-tune your acknowledgement timeouts.

PRAGMA replication_get_master

```
PRAGMA replication_get_master
```

Returns a host/port pair representing the current master. If replication has not yet started, or if the master is currently not known (such as during an election), `NULL` is returned.

PRAGMA replication_initial_master

```
PRAGMA replication_initial_master=ON|OFF
```

Causes the local environment to start up as a master site. This PRAGMA must be used once and only once in the replicated lifetime of a BDB SQL environment.

This PRAGMA is usually invoked for the first site in a new replication group before the `replication` PRAGMA is invoked and before BDB SQL initially creates the underlying BDB environment for a SQL database. Starting replication on the initial master site establishes the new replication group so that other sites can join it.

However, you must call this PRAGMA when enabling replication for a database that already exists. Doing so causes the existing database to become the replication master for a new replication group.

Note that subsequent election activity can cause other sites in the replication group to become master. Do not assume that the initial master site will remain master indefinitely, or that it will rejoin the replication group as master after a shutdown.

PRAGMA replication_local_site

```
PRAGMA replication_local_site="hostname:port"
```

Sets the local site information for replication.

PRAGMA replication_num_sites

```
PRAGMA replication_num_sites
```

Returns the number of sites in the replication group, or `0` if replication has not yet started. Any input provided to this PRAGMA is ignored.

PRAGMA replication_perm_failed

```
PRAGMA replication_perm_failed
```

Returns the number of times that a permanent message has failed the replication group's acknowledgement policy since the last time this PRAGMA was called. Zero is returned if replication has not yet started.

Every time a permanent message has failed acknowledgement, a transaction was not considered durable and the master waited the full acknowldgement timeout period before returning from its commit. If you are seeing a high number of permanent message acknowledgement failures, then that might represent a significant reduction in your application's write-throughput.

To fix the problem, begin by examining your replicas to make sure they are operating correctly, as well as your network to make sure its performance is what you expect. If all looks well, then you should either reduce your acknowledgement policy (and accept a lessened durability guarantee) or increase your acknowledgement timeout period (which might reduce your write-throughput).

To manage your acknowledgement policies, see PRAGMA replication_ack_policy (page 34). To manage your acknowledgement timeout policy, see PRAGMA replication_ack_timeout (page 35).

PRAGMA replication_priority

```
PRAGMA replication_priority=<non-0 positive integer>
```

Sets the site's election priority. When holding elections, if more than one site has the most up-to-date copy of the data then the site with the highest priority will become master. This PRAGMA may be called at any time during the life of the application. The default is 100.

If no priority is provided, then this PRAGMA returns the site's current priority.

PRAGMA replication_remote_site

```
PRAGMA replication_remote_site="hostname:port"
```

Sets information about a remote helper site in the replication group.

This PRAGMA is needed when a site first joins an existing replication group to specify a site that is already in the replication group. It must be invoked before the `replication` PRAGMA is invoked. This PRAGMA is not needed on the initial master site or when restarting a site

that is already a member of the replication group. However, supplying this PRAGMA in those situations does no harm.

Note that the information provided to this PRAGMA can be superseded by normal replication activity over the course of the environment's lifetime.

PRAGMA replication_remove_site

```
PRAGMA replication_remove_site="hostname:port"
```

Removes the specified site from the replication group. Use this PRAGMA if you truly want to remove the site permanently from the group. It is not desirable to call this PRAGMA if a site has been temporarily shut down or disconnected from the rest of the replication group.

Removing a site from the replication group means that the site is no longer counted towards the total number of sites belonging to the group. This is important when the replication group requires knowledge about whether a quorum has been reached (such as when, for example, elections are held).

PRAGMA replication_site_status

```
PRAGMA replication_site_status
```

Returns whether the local site is the `MASTER`, `CLIENT`, or `UNKNOWN`.

`UNKNOWN` is returned if replication has not yet started. CLIENT is another term for replica.

PRAGMA replication_verbose_output

```
PRAGMA replication_verbose_output=ON|OFF
```

If set to TRUE, additional logging information specifically related to replication is created.

PRAGMA replication_verbose_file

```
PRAGMA replication_verbose_file="filename"
```

Indicates that verbose replication output should be sent to the specified file, as opposed to STDOUT.

Displaying Replication Statistics

You can display a brief summary of replication statistics using `.stat :rep:`. This command displays the most basic information about replication status, as well as some information that is useful for troubleshooting. For more detailed information, use `PRAGMA statistics = REP`.

```
dbsql> .stat :rep:
Replication summary statistics
Environment configured as a replication client
Startup complete
1/50232 Maximum permanent LSN
2       Number of environments in the replication group
0       Number of failed message sends
```

```
0       Number of messages ignored due to pending recovery
0       Number of log records currently queued
```

In the above output:

- `Environment configured as a replication client`

  Identifies the current role of the site within the replication group. In this example, the current site is not the master site. *Replication client* is another term for *replica*.

- `Startup complete`

  Indicates that this replica site has completed its synchronization with the master site. Replica synchronization can take some time if there are many master transactions with which it needs to catch up.

- `Maximum permanent LSN`

  Identifies the most recent log record that is durably replicated on a master or acknowledged by a replica. You can compare a replica's maximum permanent LSN to the master's maximum permanent LSN to determine if the replica is caught up with the master.

- `Number of failed message sends`

  If this number is increasing, it could be an indication of network or communications problems between sites in the replication group.

- `Number of messages ignored due to pending recovery`

  If this number is increasing, this site is ignoring messages because it is starting up or recovering and may need some time to catch up with the rest of the replication group.

- `Number of log records currently queued`

  If this number is increasing, it means that connections to other sites may be unavailable or congested and that there may be delays in durably replicating master transactions.

Replication Usage Examples

In this section we provide two examples of using replication with BDB SQL. The first example shows a typical startup process. The second demonstrates master site failover.

Example 1: Distributed Read at 3 Sites

This example shows how a typical replication group startup sequence is performed. It initially populates the master, then starts two replicas so that read operations can be distributed. Then it shows what happens when an attempt is made to write to a replica.

Site 1:

```
# Start initial master.
# univ.db does not yet exist.
```

```
dbsql univ.db
     pragma replication_local_site="site1:7000";
     pragma replication_initial_master=ON;
     pragma replication=ON;

    # Create and populate university and country tables.
    .read university.sql
```

Site 2:

```
# Start first replica.
# univ.db does not yet exist.
dbsql univ.db
     pragma replication_local_site="site2:7001";
     pragma replication_remote_site="site1:7000";
     pragma replication=ON;
```

Site 3:

```
# Start second replica.
# univ.db does not yet exist.
dbsql univ.db
     pragma replication_local_site="site3:7002";
     pragma replication_remote_site="site1:7000";
     pragma replication=ON;
```

Site 1:

```
    # Perform some writes and reads on master.
    insert into country values ("Greenland","gl", 0, 0, 0, 2);
    insert into university values (26, "University College London",
        "ucl.edu", "uk", "Europe", 18, 39, 47, 30);
    select * from country where abbr = "gl";
    update country set top_1000 = 1 where abbr = "gl";
```

Site 2:

```
    # Perform some reads on first replica.
    select * from university where region = "Europe";
    select count(*) from country where top_100 > 0;

    # Attempt to write on first replica.
    insert into country values ("Antarctica","an", 0, 0, 0, 0);
    .../univ.db: DBcursor->put: attempt to modify a read-only database
    Error: attempt to write to a readonly database
```

Example 2: 2-Site Failover

This example demonstrates failover of the master from one site to another. It shows how a failed site can rejoin the replication group, and it shows that there is a window of time during which write operations cannot be performed for the replication group. Finally, it shows how to check the master's location.

Site 1:

```
# Start initial master.
# quote.db does not yet exist.
dbsql quote.db
     pragma replication_local_site="site1:7000";
     pragma replication_initial_master=ON;
     pragma replication=ON;

    # Create stock quote application table.
    create table stock_quote (company_name text(40), price real);
```

Site 2:

```
# Start replica.
# quote.db does not yet exist.
dbsql quote.db
     pragma replication_local_site="site2:7001";
     pragma replication_remote_site="site1:7000";
     pragma replication=ON;
```

Site 1:

```
    # Perform some writes on master.
    insert into stock_quote values ("General Electric", 20.25);
    insert into stock_quote values ("Nabisco", 24.75);
    insert into stock_quote values ("United Healthcare", 31.00);
    update stock_quote set price=25.25 where company_name = "Nabisco";
```

Site 2:

```
    # Perform some reads on replica.
    select * from stock_quote where price < 30.00;
    select price from stock_quote where
           company_name = "General Electric";
```

Site 1:

```
    # Stop the initial master.
    .exit
```

Site 2:

```
    ##########
    ### Now the remaining site does not accept write operations until
    ### the other site rejoins the replication group.
    ##########
    insert into stock_quote values ("Prudential", 17.25);
    .../quote.db: DBcursor->put: attempt to modify a read-only database
    Error: attempt to write to a readonly database
```

Site 1:

```
# Restart site, will rejoin replication group.
```

```
dbsql quote.db
    # The earlier replication=ON causes replication to be
    # automatically started.  This site may or may not become master
    # after rejoining replication group.  Check status of site's
    # startup and determine whether it is a master or a replica.
    .stat :rep:
    Replication summary statistics
    Environment configured as a replication master
    1/49056 Maximum permanent LSN
    2       Number of environments in the replication group
    0       Number of failed message sends
    0       Number of messages ignored due to pending recovery
    0       Number of log records currently queued

    # Assuming this site became master, perform some writes.
    # If this site is not the master, these writes will not
    # succeed and must be performed at the other site.
    insert into stock_quote values ("Raytheon", 9.25);
    insert into stock_quote values ("Cadbury", 7.75);
```

Site 2:

```
    # Read operations can be performed on master or replica
    # site.
    select * from stock_quote where price < 21.00;
```

Chapter 6. Administrating Berkeley DB SQL Databases

This chapter provides administrative procedures that are unique to the Berkeley DB SQL interface.

Backing Up Berkeley DB SQL Databases

You can use the standard SQLite `.dump` command to backup the data managed by the BDB SQL interface.

The BDB SQL interface supports the standard SQLite Online Backup API. However, there is a small difference between the two interfaces. In the BDB SQL interface, the value returned by the `sqlite3_backup_remaining` method and the number of pages passed to the `sqlite3_backup_step` method, are estimates of the number of pages to be copied and not exact values. To be certain that the backup process is complete, check if the `sqlite3_backup_step` method has returned `SQLITE_DONE`. To learn how to use SQLite Online Backup API, see the official SQLite Documentation Page.

Backing Up Replicated Berkeley DB SQL Databases

When BDB SQL interface databases are replicated .dump and the SQLite Online Backup API are not sufficient to create a backup. In a separate process you should use the db_hotbackup utility or other methods described in the Oracle Berkeley DB Backup Procedures section. You must then copy some additional files for the backup to be complete.

The additional files can be found in the journal directory of the source database, and should be copied into the journal directory of the backup copy. The journal directory is automatically created when a Berkeley DB SQL interface database is created. The journal directory is created in the same directory as the database file, it has the name of the database file with a `-journal` appendix.

The files that need to be copied into the backup journal directory are:

- `__db.rep.egen`
- `__db.rep.gen`
- `__db.rep.init`
- `__db.rep.system`
- `pragma`

Syncing with Oracle Databases

Oracle's SQLite Mobile Client product allows you to synchronize a SQLite database with a back-end Oracle database. Because the BDB SQL interface is a drop-in replacement for SQLite, this means you can synchronize a Berkeley DB database with an Oracle back-end as well.

Note

Berkeley DB SQL databases are not compatible with SQLite databases. In order for sync to work, you must remove any currently existing SQLite databases.

Syncing on Unix Platforms

For Unix platforms, the easiest way to use Oracle's SQLite Mobile Client is to build the BDB SQL interface with the compatibility option. That is, specify both `--enable-sql` and `--enable-sql-compat` when you configure your Berkeley DB installation. This causes libraries with the exact same name as the SQLite libraries to be created when you build Berkeley DB.

Having done that, you must then change your platform's library search path so that it finds the Berkeley DB libraries *before* any installed SQLite libraries. On many (but not all) Unix platforms, you do this by modifying the `LD_LIBRARY_PATH` environment variable. See your operating system documentation for information on how to change your search path for dynamically linked libraries.

Once you have properly configured and built your Berkeley DB installation, and you have properly configured your operating system, you can use the Oracle SQLite Mobile Client in exactly the same way as you would if you were using standard SQLite libraries and databases with it. See the Oracle Database Mobile Server documentation. documentation for information on using SQLite Mobile Client.

For information on building the BDB SQL interface, see the Configuring the SQL Interface section in the *Berkeley DB Installation and Build Guide*.

Syncing on Windows Platforms

For Windows platforms, you use Oracle's SQLite Mobile Client by building the BDB SQL interface in the same way as you normally do. See the Building Berkeley DB for Windows chapter in the *Berkeley DB Installation and Build Guide* for more information.

Once you have built the product, rename the Berkeley DB SQL dlls so that they are named identically to the standard SQLite dlls (sqlite3.dll). Install the renamed Berkeley DB SQL dll along with the main Berkeley DB dll (libdb5x.dll) in the same directory as the SQLite dlls. See the Building the SQL API section for details.

Finally, configure your Windows PATH environment variable so that it finds your Berkeley DB dlls before it finds any standard SQLite dlls that might be installed on your system.

Once you have built your Berkeley DB installation and renamed your dlls, and you have properly configured your operating system, you can use the Oracle SQLite Mobile Client in exactly the same way as you would if you were using standard SQLite libraries and databases with it. See the Oracle Database Mobile Server documentation for information on using SQLite Mobile Client.

Syncing on Windows Mobile Platforms

For Windows Mobile platforms, you use Oracle's SQLite Mobile Client by building the BDB SQL interface in the same way as you normally do. See the Building Berkeley DB for Windows Mobile chapter in the *Berkeley DB Installation and Build Guide* for more information.

Once you have built the product, rename the Berkeley DB SQL dll to `sqlite3.dll`. Then, copy the dll to the `\Windows` path on the phone. Note that you only need the new `sqlite3.dll`; you do not need any of the other Berkeley DB dlls.

Once you have built your Berkeley DB installation and renamed your dlls, and you have properly configured your operating system, you can use the Oracle SQLite Mobile Client in exactly the same way as you would if you were using standard SQLite libraries and databases with it. See the Oracle Database Mobile Server documentation for information on using SQLite Mobile Client.

Data Migration

If you have a database created by SQLite, you can migrate it to a Berkeley DB database for use with the BDB SQL interface. For production applications, you should do this only when your application is shutdown.

All data and schema supported by SQLite can be migrated to a Berkeley DB database.

Migration Using the Shells

To migrate your data from SQLite to a Berkeley DB database:

1. Make sure your application is shutdown.
2. Open the SQLite database within the **sqlite3** shell.
3. Execute the `.output` command to specify the location where you want to dump data.
4. Dump the database using the SQLite `.dump` command.
5. Close the **sqlite3** shell and open the Berkeley DB dbsql shell.
6. Load the dumped data using the `.read` command.

Note that you can migrate in the reverse direction as well. Dump the Berkeley DB database by calling `.dump` from within the Berkeley DB dbsql shell, and load it into SQLite by `.read` from within SQLite's **sqlite3** shell.

Catastrophic Recovery

Catastrophic recovery is used when normal recovery is unable to repair a corrupted database. A detailed description of the recovery utility is described at db_recover . This section describes how to use the utility with a SQL database.

The first step in running recovery is to add a DB_CONFIG file to the journal directory. In the file, add the following line:
`add_data_dir ..`

Then execute the following command in the journal directory **db_recover -c**.

Database Statistics

The statistics utility can be used to print out information on tables in a database. A detailed description of the statistics utility is described at db_stat . This section describes how to use the utility with a SQL database.

The first step in getting statistics on a table is to find the internal name for the table. The internal table name is `table#####`, where the number is the rootpage number, which can be retrieved using the following SQL statement:
`SELECT rootpage FROM sqlite_master WHERE name='[table name]';` For example, if the root page is 10, the internal table name will be `table00010`.

In order to run the **db_stat** utility, first add a DB_CONFIG file to the journal directory. In the file, add the following line:
`add_data_dir ..`

Then execute the following command in the directory with the database: **db_stat -d [file name] -h [journal directory] -s [internal table name]**. That command will print out information on the table.

Verify Database Structure

Verification is used to confirm that a database structure is correct, and has not been corrupted by a crash or other failure. A detailed description of the verify utility is described at db_verify . This section describes how to use the utility with a SQL database.

The first step in running verify is to add a DB_CONFIG file to the journal directory. In the file, add the following line:
`add_data_dir ..`

Then execute the following command in the database directory **db_verify -h [journal directory] [database file]**. The command will either print out a message reporting success, or will list any error found in the database.

Appendix A. Using the BFILE Extension

The BFILE data type allows the BDB SQL interface to access binary files that are stored in the file system outside of the database. The binary file can be queried in exactly the same way as any other data type stored in the database, but Berkeley DB is able to save space in the database file by not embedding a large amount of binary data in it. This also helps overall database performance.

Internally, a BFILE column or attribute stores a BFILE locater, which serves as a pointer to the binary file. The locater maintains the directory alias and the filename. You can change the path of BFILE without affecting the base table by using the BFILENAME function. BFILE is somewhat like the BLOB data type, but it does not participate in transactions and it is not recoverable. Instead, the underlying operating system is expected to provide file integrity and durability.

The remainder of this section describes the various objects and functions that the BFILE extension makes available to you. In addition, complete examples of using these extensions are available with your Berkeley DB distribution. They are placed in the following location:

```
<db-dist>/lang/sql/sqlite/ext/bfile/examples
```

Supported Platforms and Languages

The BFILE extension is currently only supported for *nix platforms.

The BFILE extension it is not available in your library by default. Instead, you must enable the extension when you compile Berkeley DB. See the *Berkeley DB Installation and Build Guide* for information on how to enable this extension when you build Berkeley DB. Once you have enabled the extension, applications will also need to load the BFILE library file: `libbfile_ext.so`.

By default, the BFILE extension provides support for additional SQL statements. With some extra configuration at Berkeley DB compile time, you can also obtain support for extensions to the SQLite C/C++ interface. Both the SQL extensions and the extensions to the SQLite C/C++ interface are described in the following sections.

BFILE SQL Objects and Functions

When the BFILE extension is enabled, you can create a `DIRECTORY` object. These objects are required before you can store a pointer to a file in a `BFILE` column.

`DIRECTORY` objects are stored in a special table called `BFILE_DIRECTORY`. This table is automatically created for you when it is needed. You should *not* manually create this table.

You manage `DIRECTORY` objects using the following SQL functions:

BFILE_CREATE_DIRECTORY (page 47)

The following sections describe the SQL functions that you can use when the BFILE extension is enabled.

BFILE_CREATE_DIRECTORY

```
BFILE_CREATE_DIRECTORY(directory, path)
```

Creates a `DIRECTORY` object as a path. The specified path must not already exist, or `Directory already exists` is returned.

BFILE_REPLACE_DIRECTORY

```
BFILE_REPLACE_DIRECTORY(directory, path)
```

Replaces the named `DIRECTORY` object using the specified path. If the object does not exist, `Directory does not exist` is returned.

BFILE_DROP_DIRECTORY

```
BFILE_DROP_DIRECTORY(directory)
```

Drops the named `DIRECTORY` object. If the object does not exist, `Directory does not exist` is returned.

BFILE_NAME

```
BFILE_NAME(directory, filename)
```

Returns the BFILE locator.

BFILE_FULLPATH

```
BFILE_FULLPATH(column)
```

Returns the full path.

BFILE_OPEN

```
BFILE_OPEN(column)
```

Extracts the directory and file names from the BFILE locator, and then opens that file. On success, a BFILE handle is returned. Otherwise, `0` is returned.

BFILE_READ

```
BFILE_READ(BFILE handle, amt, offset)
```

Reads at most `amt` data from the BFILE handle, starting at `offset`. On success, `Data` is returned. Otherwise, `0` is returned to indicate that no more valid data is available.

BFILE_CLOSE

```
BFILE_CLOSE(BFILE handle)
```

Closes the BFILE handle.

BFILE_SIZE

```
BFILE_SIZE(column)
```

Returns the size of the BFILE. On success, the size is returned. Otherwise, -1 is returned.

BFILE C/C++ Objects and Functions

The BFILE extension can optionally make available to you some additional C language data types and functions for use with the SQLite C/C++ interface. These are available to you only if you take the proper steps when you compile Berkeley DB. See the *Berkeley DB Installation and Build Guide* for more information.

Once enabled, the BFILE C extension makes the following new structure available to you:

```
typedef struct sqlite3_bfile sqlite3_bfile;
```

This structure serves as the BFILE handle when you are using the BFILE extension along with the SQLite C/C++ interface.

In addition to the new structure, you can also use the following new C functions:

sqlite3_column_bfile

```
int
sqlite3_column_bfile(sqlite3_stmt *pStmt, int iCol,
                     sqlite3_bfile **ppBfile);
```

Returns a result set from a query against a column of type BFILE.

On success, `SQLITE_OK` is returned and the new BFILE handle is written to **ppBfile**. Otherwise, `SQLITE_ERROR` is returned.

Parameters are:

- **pStmt**

  Pointer to the prepared statement that the function is evaluating. The statement is created using `sqlite3_prepare_v2()` or one of its variants.

  If this statement does not point to a valid row, the result is undefined.

- **iCol**

  Index of the column for which information should be returned. The left-most column of the result set is index `0`. Use `sqlite3_column_count()` to discover the number of columns in the result set.

  If the column index is out of range, the result is undefined.

- **ppBfile**

The BFILE handle that you are using for the query. This pointer is valid only until `sqlite3_step()`, `sqlite3_reset()` or `sqlite3_finalize()` have been called.

The memory space used to hold this handle is freed by sqlite3_bfile_final (page 51) Do not pass these pointers to `sqlite3_free()`.

This function can be called successfully only if all of the following conditions are true. If any of the following are not true, the result is undefined:

- The most recent call to `sqlite3_step()` has returned `SQLITE_ROW`.
- Neither `sqlite3_reset()` nor `sqlite3_finalize()` have been called since the last time `sqlite3_step()` was called.
- `sqlite3_step()`, `sqlite3_reset()` or `sqlite3_finalize()` have not been called from a different thread while this routine is pending.

sqlite3_bfile_open

```
int
sqlite3_bfile_open(sqlite3_bfile *pBfile);
```

Opens a file for incremental read.

On success, `SQLITE_OK` is returned. Otherwise, `SQLITE_ERROR` is returned.

To avoid a resource leak, every opened BFILE handle should eventually be closed with the sqlite3_bfile_close (page 49) function. Note that **pBfile** is always initialized such that it is always safe to invoke `sqlite_bfile_close()` against it, regardless of the success or failure of this function.

sqlite3_bfile_close

```
int
sqlite3_bfile_close(sqlite3_bfile *pBfile);
```

Closes an open BFILE handle. The BFILE is closed unconditionally. Even if this function returns an error, the BFILE is still closed.

Calling this routine with a null pointer (such as would be returned by failed call to sqlite3_column_bfile()) is a harmless non-operation.

On success, `SQLITE_OK` is returned. Otherwise, `SQLITE_ERROR` is returned.

sqlite3_bfile_is_open

```
int
sqlite3_bfile_is_open(sqlite3_bfile *pBfile, int *open);
```

Checks whether a BFILE handle is open. The open parameter is set to 1 if the file is open, otherwise it is 0.

On success, SQLITE_OK is returned. Otherwise, SQLITE_ERROR is returned.

sqlite3_bfile_read

```
int
sqlite3_bfile_read(sqlite3_bfile *pBfile, void *oBuff, int nSize,
                   int iOffset, int *nRead);
```

This function is used to read data from an opened BFILE handle into a caller-supplied buffer.

On success, SQLITE_OK is returned, the data that has been read is written to the output buffer, **oBuff**, and the amount of data written to the buffer is recorded in **nRead**. Otherwise, SQLITE_ERROR is returned.

Parameters are:

- **pBfile**

  The BFILE handle from which the data is read.

  This function only works on a BFILE handle which has been created by a prior successful call to sqlite3_bfile_open (page 49) and which has not been closed by sqlite3_bfile_close (page 49). Passing any other pointer in to this function results in undefined and probably undesirable behavior.

- **oBuff**

  The buffer used to contain the data that is read from **pBfile**. It must be at least **nSize** bytes in size.

- **nSize**

  The amount of data, in bytes, to read from the BFILE.

- **iOffset**

  The offset from the beginning of the file where the read operation is to begin.

- **nRead**

  Contains the amount of data, in bytes, actually written to buffer **oBuff** once the read operation is completed.

  ### Note

  The size of the BFILE can be determined using the sqlite3_bfile_size (page 51) function.

sqlite3_bfile_file_exists

```
int
sqlite3_bfile_file_exists(sqlite3_bfile *pBfile, int *exist);
```

Checks whether a BFILE exists. The `exists` parameter is set to 1 if the file is exists, otherwise it is 0.

On success, `SQLITE_OK` is returned. Otherwise, `SQLITE_ERROR` is returned.

sqlite3_bfile_size

```
int
sqlite3_bfile_size(sqlite3_bfile *pBfile, off_t *size);
```

Returns the size of the BFILE, in bytes.

On success, `SQLITE_OK` is returned, and **size** is set to the size of the BFILE, in bytes. Otherwise, `SQLITE_ERROR` is returned.

This function only works on a BFILE handle which has been created by a prior successful call to sqlite3_column_bfile (page 48) and which has not been finalized by sqlite3_bfile_final (page 51). Passing any other pointer in to this function results in undefined and probably undesirable behavior.

sqlite3_bfile_final

```
int
sqlite3_bfile_final(sqlite3_bfile *pBfile);
```

Frees a BFILE handle.

On success, `SQLITE_OK` is returned. Otherwise, `SQLITE_ERROR` is returned.