Oracle Berkeley DB

Getting Started with the SQL APIs

11g Release 2

ORACLE®

BERKELEY DB

Legal Notice

Published 6/25/2010

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 11*g* Release 2. 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

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 (page 15), 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.

How you install the BDB SQL interface differs depending on whether you are using a Unix or a Windows system.

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 `dlls` 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_sql50.dll`

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

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` 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.

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 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.

For more information on the journal directory, see Introduction to Environments (page 9).

Unsupported PRAGMAs

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

PRAGMA auto_vacuum
PRAGMA incremental_vacuum
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.

PRAGMA journal_size_limit

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

Berkeley DB does not have a journal file, but it does write and use *log files*. Over the course of the database's lifetime, Berkeley DB will probably create multiple log files. A new log file is created when the current log file has reached the defined maximum size for a log file.

You use `PRAGMA journal_size_limit` to define this maximum size for a log file.

For more information, see Setting the Log File Size (page 12).

PRAGMA max_page_count

For standard SQLite, this identifies the maximum number of pages allowed in the database. For the BDB SQL interface, this identifies the maximum size (in bytes) that the database file is allowed to be.

For both interfaces, this pragma performs essentially the same function, but you express the upper bound in a slightly different way depending on which interface you are using.

For more information, see Configuring the Database Page Size (page 10).

Miscellaneous Differences

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

- The BDB SQL interface does not support the `EXCLUSIVE` and `IMMEDIATE` keywords. Use of `BEGIN EXCLUSIVE` and `BEGIN IMMEDIATE` cause a deferred transaction (the default type of transaction) to be started.
- 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 5) for more information.
- When you use the BDB SQL interface, you cannot attach a database more than once within the same runtime.

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. This might be interesting to you in some administrative situations. For some minimal information on what an environment is, see Introduction to Environments (page 9).
- The Locking Subsystem

  You can configure the maximum number of locks that can be in use at any given time when you use the BDB SQL interface. This is probably only interesting to you if you are using the BDB SQL interface in a concurrent application that is running a very large number of transactions.

  For information on configuring your locking subsystem, see Managing the Locking Subsystem (page 14).
- The Logging Subsystem

  The BDB SQL interface maintains log files in its journal directory, and you can manage various aspects of these. For the overwhelming majority of applications, there is no need to manage this. But for the sake of completeness, this topic is described in this manual.

  For more information, see Administering Log Files (page 12).

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.

Under the hood, 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 DB SQL Interface

When you use the BDB SQL interface, the lock usage is considerably different. First, you cannot specify the kind of a lock that you want when you begin a transaction. That is, the `IMMEDIATE` and `EXCLUSIVE` keywords are ignored by the BDB SQL interface. Instead, you simply begin your transaction with `BEGIN`.

Note that this does not mean that Berkeley DB only supports one kind of a lock. Instead, 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. What you do not have to do (and, in fact, cannot do), is identify the type of lock that you want when you begin your transaction.

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);
Error: database table is locked
```

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.

The proper thing for your application to do here is to rollback the transaction for one of the sessions and then retry the operation. This is exactly what you would do if you were using SQLite in shared cache mode.

Chapter 3. Configuring 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. 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 for lines of the format **NAME VALUE** when your environment is opened. This happens when your application starts up and creates its first connection to the database.

One or more whitespace characters are used to delimit the two parts of the line, and trailing whitespace characters are discarded. All empty lines or lines whose first character is a whitespace or hash (#) character are ignored. Each line must specify both the NAME and the VALUE of the pair. The specific NAME VALUE pairs you can use with the BDB SQL interface are documented in DB_CONFIG Parameter Reference (page 22).

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 Environment Creation

In order to provide the `DB_CONFIG` file before the environment is first created, physically make the environment 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.

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:

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

Configuring the Database 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. If you adjust the database page size, you may also want to adjust the in-memory cache size, as described in Configuring the In-Memory Cache (page 11).

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.

Selecting the Page Size

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 5) 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*.

Selecting the Database File Size

Berkeley DB sets an upper bound on how large your database file size is allowed to be. Any attempt to insert data into the database that grows this file beyond this upper bound results in a failure.

You can set the upper bound for your database file size using `PRAGMA max_page_count`. Issue this PRAGMA with no value to see what the current maximum database file is.

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.

Note that you can change the cache size only if no table operations have been executed on the database. In other words, to change your cache size:

- Open a handle to your database.
- Execute `PRAGMA cache_size`
- Proceed with any table modification operations (`CREATE`, `UPDATE`, `INSERT`, `SELECT`) that you might want to perform.

Alternatively, you can set you cache size with your `DB_CONFIG` file, and so skip the necessity of executing the PRAGMA. See set_cachesize (page 22) for details.

Administering Log Files

Your environment directory contains log files. Berkeley DB log files are used to record all the transactional activity performed against the Berkeley DB database files. This information is used after an application or system failure to automatically restore the database to an up-to-date consistent point.

Your log files are maintained by Berkeley DB's logging subsystem. There are some aspects of the Berkeley DB logging subsystem that you can configure using `DB_CONFIG` parameters, and (sometimes) by using PRAGMAs.

Note

For most users of the BDB SQL interface, there should not normally be any reason to manage your log files or otherwise worry about them. However, it is important to realize that they can not simply be deleted. Note that when using the Berkeley DB SQL interface, your log files will be automatically deleted by Berkeley DB when they are no longer needed.

The things you can manage for your logging subsystem are:

- Size of the log files. See Setting the Log File Size (page 12).
- Size of the logging subsystem's region. See Configuring the Logging Region Size (page 13).
- Size of the log buffer in memory. Setting the In-Memory Log Buffer Size (page 13).

Setting the Log File Size

Whenever a pre-defined amount of data is written to a log file (10 MB by default), the BDB SQL interface stops using the current log file and starts writing to a new file. You can change the maximum amount of data contained in each log file by using either `PRAGMA journal_size_limit` or the `set_lg_max` DB_CONFIG file parameter.

If you use `PRAGMA journal_size_limit`, then using this PRAGMA with no value simply returns the current journal size limit. Using:

```
PRAGMA journal_size_limit=N
```

sets the log size to *N* bytes. If the PRAGMA is successful, *N* is returned. If it fails, the previous log file size is returned. Failures can occur if you specify a log file size that is less than 4K bytes, or if you specify a log file size larger than the permitted file size on the system.

If you use the DB_CONFIG file to manage this value, `set_lg_max` may be changed without re-creating the environment. You will, however, have to restart your application in order for the DB_CONFIG file to be re-read.

The DB_CONFIG file is described in The DB_CONFIG File (page 9). The `set_lg_max` parameter is described in set_lg_max (page 24).

For a description of how, when and why you should change the size of your log files, see the Selecting a page size section in the *Berkeley DB Programmer's Reference Guide*.

Configuring the Logging Region Size

The logging subsystem's default region size is 512 KB. The logging region is used to store database and table names, and so you may need to increase its size if you will be using a large number of tables.

You can set the size of your logging region by using the `set_lg_regionmax` DB_CONFIG parameter. Note that to manage this value you must set it before you create your environment, or you must re-create your environment. See The DB_CONFIG File (page 9) for more information.

The `set_lg_regionmax` parameter is described in set_lg_regionmax (page 25).

Setting the In-Memory Log Buffer Size

When using named (persistent) databases, log information is stored in-memory until the storage space fills up, or a transaction commit forces the log information to be flushed to disk.

It is possible to increase the amount of memory available to your file log buffer. Doing so improves throughput for long-running transactions, or for transactions that produce a large amount of data. Note that for named (persistent) databases, the default log buffer space is 32 KB.

You can increase your log buffer space by using the `set_lg_bsize` DB_CONFIG parameter. Note that this method can only be called before the environment is first opened, so you will have to set this by creating your `-journal` directory, and then creating your database. See The DB_CONFIG File (page 9) for more information.

The `set_lg_bsize` parameter is described in set_lg_bsize (page 24).

Note

When working with in-memory databases, the environment is configured to perform logging in-memory. The log buffer is set to 64 * 1024, and the maximum log size is set to 32 * 1024.

Managing the Locking Subsystem

Whenever the BDB SQL interface reads from or writes to the database, the underlying Berkeley DB code must acquire locks. These locks represent a finite resource. For most installations, you should never have to worry about the locking resources available to Berkeley DB because the default values are appropriate for most applications.

However, if your application is using an extremely large number of threads that are all simultaneously accessing your data, then you might have to increase your locking resources. Similarly, if your database contains a very large number of tables that you are accessing using one or more simultaneous threads or processes, then you might also need to increase your locking resources.

On the other hand, if you are using the BDB SQL interface on devices with extremely limited resources, then you might want to reduce your locking resources.

All of these values must be configured before your environment is first created. To change these values after environment creation time, you must re-create the environment. See The DB_CONFIG File (page 9) for more information.

The maximum locking values that you can manage, and the `DB_CONFIG` parameter that you use to manage that value, are:

- The maximum number of lockers supported by the environment. This value is used by the environment when it is opened to estimate the amount of space that it should allocate for various internal data structures. By default, 2,000 lockers are supported.

  The maximum number of lockers corresponds roughly to the maximum number of concurrent transactions in the system.

  To configure this value, use the `set_lk_max_lockers` DB_CONFIG parameter. See set_lk_max_lockers (page 23) for details.

- The maximum number of locks supported by the environment. By default, 10,000 locks are supported.

  To configure this value, use the `set_lk_max_locks` DB_CONFIG parameter. See set_lk_max_locks (page 23) for details.

- The maximum number of locked objects supported by the environment. By default, 10,000 objects can be locked.

  To configure this value, use the `set_lk_max_objects` DB_CONFIG parameter. See set_lk_max_objects (page 23) for details.

Note that when you are using the BDB SQL interface, the default values provided in the previous list are different from the default values used by Berkeley DB in general. For Berkeley DB in general, the defaults for all these values are set to 1,000.

Chapter 4. Administrating Berkeley DB SQL

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

Backing up

You can use the standard SQLite `.dump` command to backup the data managed in by the BDB SQL interface. You can also use the standard Berkeley DB backup mechanisms on the database. This section describes the mechanisms that can be performed from the command line.

Note

The standard SQLite online backup API is not supported by the BDB SQL interface.

Offline Backups

To create an offline backup:

1. Commit or abort all on-going transactions.
2. Pause all database writes.
3. Force a checkpoint. See the db_checkpoint command line utility.
4. Copy your database file to the backup location. Note that in order to perform recovery from this backup, do not change the name of the database file.
5. Copy the *last* log file to your backup location. Your log files are named `log.`*xxxxxxxxxx*, where *xxxxxxxxxx* is a sequential number. The last log file is the file with the highest number.

   Remember that your log files are placed in the environment directory, which is created on-disk next to your database file. It has the same name as your database file, but adds a `-journal` extension. For example, if your database is named `mydb.db`, then your environment directory is named `mydb.db-journal`

Hot Backup

To create a hot backup, you do not have to stop database operations. Transactions may be on-going and you can be writing to your database at the time of the backup. However, this means that you do not know exactly what the state of your database is at the time of the backup.

You can use the db_hotbackup command line utility to create a hot backup for you. This utility will (optionally) run a checkpoint and then copy all necessary files to a target directory. To do this when you are using the BDB SQL interface:

1. Create a `DB_CONFIG` file in your environment directory.
2. Add a `set_data_dir` parameter to the `DB_CONFIG` file. This parameter indicates what directory contains the actual Berkeley DB database managed by this environment. That directory is one level up from you environment, so you want this parameter to be:

```
set_data_dir ..
```

3. Add a `setl_lg_dir` parameter to the `DB_CONFIG` file. This parameter identifies the directory that contains the environment's log files. This parameter should be:

   ```
   set_lg_dir .
   ```

4. Run the db_hotbackup command:

   ```
   db_hotbackup -h [environment directory] -b [target directory] -D
   ```

   The `-D` option tells the utility to read the `DB_CONFIG` file before running the backup.

Alternatively, you can manually create a hot backup as follows:

1. Copy your database file to the backup location. Note that in order to perform recovery from this backup, do not change the name of the database file.
2. Copy all logs to your backup location.

   Remember that your log files are placed in the environment directory.

Note

It is important to copy your database file *and then* your logs. In this way, you can complete or roll back any database operations that were only partially completed when you copied the database.

Incremental Backups

Once you have created a full backup (that is, either a offline or hot backup), you can create incremental backups. To do this, simply copy all of your currently existing log files to your backup location.

Incremental backups do not require you to run a checkpoint or to cease database write operations.

When you are working with incremental backups, remember that the greater the number of log files contained in your backup, the longer recovery will take. You should run full backups on some interval, and then do incremental backups on a shorter interval. How frequently you need to run a full backup is determined by the rate at which your database changes and how sensitive your application is to lengthy recoveries (should one be required).

You can also shorten recovery time by running recovery against the backup as you take each incremental backup. Running recovery as you go means that there will be less work for the BDB SQL interface to do if you should ever need to restore your environment from the backup.

About Unix Copy Utilities

If you are copying database files you must copy databases atomically, in multiples of the database page size. In other words, the reads made by the copy program must not be interleaved with writes by other threads of control, and the copy program must read the databases in multiples of the underlying database page size. Generally, this is not a problem because operating systems already make this guarantee and system utilities normally read in power-of-2 sized chunks, which are larger than the largest possible Berkeley DB database page size.

On some platforms (most notably, some releases of Solaris), the copy utility (`cp`) was implemented using the `mmap()` system call rather than the `read()` system call. Because `mmap()` did not make the same guarantee of read atomicity as did `read()`, the `cp` utility could create corrupted copies of the databases.

Also, some platforms have implementations of the `tar` utility that performs 10KB block reads by default. Even when an output block size is specified, the utility will still not read the underlying database in multiples of the specified block size. Again, the result can be a corrupted backup.

To fix these problems, use the `dd` utility instead of `cp` or `tar`. When you use `dd`, make sure you specify a block size that is equal to, or an even multiple of, your database page size. Finally, if you plan to use a system utility to copy database files, you may want to use a system call trace utility (for example, `ktrace` or `truss`) to make sure you are not using a I/O size that is smaller than your database page size. You can also use these utilities to make sure the system utility is not using a system call other than `read()`.

Recovering from a Backup

If you used standard Berkeley DB backup procedures to backup your database, then you can restore your database using the procedures described in this section.

Note that Berkeley DB supports two types of recovery:

- Normal recovery, which examines only those log records needed to bring the database to a consistent state since the last checkpoint. Normal recovery starts with any logs used by any transactions active at the time of the last checkpoint, and examines all logs from then to the current logs.

  Normal recovery is automatically run (if necessary) when you open your database. It is necessary to run recovery if a thread or process shuts down without properly closing the database.

- Catastrophic recovery examines all available log files. You use catastrophic recovery to restore your database from a previously created backup.

Catastrophic Recovery

Use catastrophic recovery when you are recovering your database from a previously created backup. Note that to restore your database from a previous backup, you should copy the backup to a new environment directory, and then run catastrophic recovery. Failure to do so can lead to the internal database structures being out of sync with your log files.

To run catastrophic recovery:

- Shutdown all database operations.

- Restore the backup to an empty directory. This means you need your database file, as well as the `-journal` directory, and any available log files that the backup contains.

  Note that the backup database file and the journal directory must have the same name as the database and journal directory that you are restoring. You can put the backup in a different location on disk, but the name of the file and directory must remain the same.

- Make sure that a `DB_CONFIG` file exists in the journal directory that you are using to restore your database. This file must contain a the following line:

  ```
  set_data_dir ..
  ```

- Run the db_recover command line utility with the `-c` option.

Note that catastrophic recovery examines every available log file — not just those log files created since the last checkpoint as is the case for normal recovery. For this reason, catastrophic recovery is likely to take longer than does normal recovery.

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 Lite 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 Lite 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 Lite 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.

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. Note that if you build the BDB SQL interface with the compatibility option, you can alternatively use Berkeley DB's sqlite3 utility.

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 dbsql shell, and load it into SQLite by calling `.read` from within SQLite's **sqlite3** shell.

Supported Data and Schema

You can migrate data between SQLite and Berkeley DB that uses the UTF-8 character encoding.

The following data types can be migrated between SQLite and Berkeley DB:

- CHAR, TEXT , VARCHAR, NVARCHAR, STRING
- REAL, DOUBLE, FLOAT
- INTEGER, BOOLEAN, BIG INTEGER, NUMBER
- NUMERIC
- BLOB, CLOB
- NULL, NOT NULL
- COLLATE BINARY, COLLATE RTRIM, COLLATE NOCASE
- DATETIME, CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP

The following schema can be migrated between SQLite and Berkeley DB:

- PRAGMA writable_schema=ON/OFF
- PRAGMA foreign_keys=ON/OFF
- PRAGMA cache_size
- CREATE TABLE
  - PRIMARY KEY
  - UNIQUE
  - CONFLICT IGNORE, FAIL, REPLACE, ABORT, ROOLBACK
  - REFERENCE ON ... CASECADE, ON ... NO ACTION, DEFERRABLE INITIALLY DEFERRED, and so forth.

  - AUTOINCREMENT
  - Static DEFAULT value, dynamic DEFAULT value
  - Functions such as datetime, typeof, and so forth.
  - ASC, DESC
  - HIDDEN
  - CHECK
- CREATE INDEX, UNIQUE INDEX
- CREATE VIEW
  - SELECT statement, ANALYZE
  - JOIN
  - UNION
- CREATE TRIGGER AFTER/BEFORE BEGIN
- CREATE VIRTUAL TABLE USING
- INSERT

Appendix A. DB_CONFIG Parameter Reference

The following `DB_CONFIG` parameters can be used to manage various aspects of your application's database environment. For information on your database environment, and how and where to set these parameters, see The DB_CONFIG File (page 9).

set_cachesize

Sets the size of the shared memory buffer pool — that is, the cache. The cache should be the size of the normal working data set of the application, with some small amount of additional memory for unusual situations. (Note: the working set is not the same as the number of pages accessed simultaneously, and is usually much larger.)

The value specified for this parameter is the *maximum* value that your application will be able to use for your in-memory cache. If your application does not have enough data to fill up the amount of space specified here, then your application will only use the amount of memory required by the data that your application does have.

For the BDB SQL interface, the default cache size is 8MB. You cannot specify a cache size value of less than 100KB.

Any cache size less than 500MB is automatically increased by 25% to account for cache overhead; cache sizes larger than 500MB are used as specified. The maximum size of a single cache is 4GB on 32-bit systems and 10TB on 64-bit systems. (All sizes are in powers-of-two, that is, 256KB is 2^18 not 256,000.)

It is possible to specify cache sizes large enough they cannot be allocated contiguously on some architectures. For example, some releases of Solaris limit the amount of memory that may be allocated contiguously by a process. If **ncache** is 0 or 1, the cache will be allocated contiguously in memory. If it is greater than 1, the cache will be split across **ncache** separate regions, where the **region size** is equal to the initial cache size divided by **ncache**.

The cache size supplied to this parameter will be rounded to the nearest multiple of the region size and may not be larger than the maximum possible cache size configured for your application (use the set_cache_max (page 23) to do this). The **ncache** parameter is ignored when resizing the cache.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_cachesize`, one or more whitespace characters, and the initial cache size specified in three parts: the gigabytes of cache, the additional bytes of cache, and the number of caches, also separated by whitespace characters. For example:

```
set_cachesize 2 524288000 1
```

creates a single 2.5GB physical cache.

Note that this parameter is ignored unless it is specified before you initially create your environment, or you re-create your environment after changing it. For more information, see The DB_CONFIG File (page 9)

set_cache_max

Sets the maximum size that the `set_cachesize` parameter is allowed to set. The specified size is rounded to the nearest multiple of the cache region size, which is the initial cache size divided by the number of regions specified to the `set_cachesize` parameter. If no value is specified, it defaults to the initial cache size.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_cache_max`, one or more whitespace characters, and the maximum cache size in bytes, specified in two parts: the gigabytes of cache and the additional bytes of cache. For example:

```
set_cache_max 2 524288000
```

sets the maximum cache size to 2.5GB.

This parameter can be changed with a simple restart of your application; you do not need to re-create your environment for it to be changed.

set_lk_max_lockers

Sets the maximum number of locking entities supported by the Berkeley DB environment. This value is used by Berkeley DB to estimate how much space to allocate for various lock-table data structures. When using the BDB SQL interface, the default value is 2,000 lockers.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_lk_max_lockers`, one or more whitespace characters, and the number of lockers.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment as described in The DB_CONFIG File (page 9).

set_lk_max_locks

Sets the maximum number of locks supported by the Berkeley DB environment. This value is used to estimate how much space to allocate for various lock-table data structures. When using the BDB SQL interface, the default value is 10,000 locks.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_lk_max_locks`, one or more whitespace characters, and the number of locks.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment as described in The DB_CONFIG File (page 9).

set_lk_max_objects

Sets the maximum number of locked objects supported by the Berkeley DB environment. This value is used to estimate how much space to allocate for various lock-table data structures. When using the BDB SQL interface, the default value is 10,000 objects.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_lk_max_objects`, one or more whitespace characters, and the number of objects.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment as described in The DB_CONFIG File (page 9).

set_lg_bsize

Sets the size of the in-memory log buffer, in bytes.

For the BDB SQL interface, when the logging subsystem is configured for on-disk logging, the default size of the in-memory log buffer is approximately 64KB. Log information is stored in-memory until the storage space fills up or a transaction commit forces the information to be flushed to stable storage. In the presence of long-running transactions or transactions producing large amounts of data, larger buffer sizes can increase throughput.

When the logging subsystem is configured for in-memory logging, the default size of the in-memory log buffer is 1MB. Log information is stored in-memory until the storage space fills up or transaction abort or commit frees up the memory for new transactions. In the presence of long-running transactions or transactions producing large amounts of data, the buffer size must be sufficient to hold all log information that can accumulate during the longest running transaction. When choosing log buffer and file sizes for in-memory logs, applications should ensure the in-memory log buffer size is large enough that no transaction will ever span the entire buffer, and avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started in the first log "file" is still active.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_lg_bsize`, one or more whitespace characters, and the log buffer size in bytes.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment as described in The DB_CONFIG File (page 9).

set_lg_max

Sets the maximum size of a single file in the log, in bytes. The value set for this parameter may not be larger than the maximum unsigned four-byte value.

When the logging subsystem is configured for on-disk logging, the default size of a log file is 10MB.

When the logging subsystem is configured for in-memory logging, the default size of a log file is 256KB. In addition, the configured log buffer size must be larger than the log file size. (The logging subsystem divides memory configured for in-memory log records into "files", as database environments configured for in-memory log records may exchange log records with other members of a replication group, and those members may be configured to store log records on-disk.) When choosing log buffer and file sizes for in-memory logs, applications should ensure the in-memory log buffer size is large enough that no transaction will ever span the entire buffer, and avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started in the first log "file" is still active.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_lg_max`, one or more whitespace characters, and the maximum log file size in bytes.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment as described in The DB_CONFIG File (page 9).

set_lg_regionmax

Set the size of the underlying logging area of the Berkeley DB environment, in bytes. By default, or if the value is set to 0, the minimum region size is used, approximately 128KB. The log region is used to store filenames, and so may need to be increased in size if a large number of files will be opened and registered with the specified Berkeley DB environment's log manager.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_lg_regionmax`, one or more whitespace characters, and the log region size in bytes.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment as described in The DB_CONFIG File (page 9).