An Oracle White Paper

Oracle Berkeley DB SQL API vs. SQLite API – Integration, Benefits and Differences

Introduction 1
The Berkeley DB Architecture 2
Berkeley DB Benefits.. 3
Berkeley DB Differences 5
    Configuration.. 6
    Contention Handling.. 8
    Tuning 9
Conclusion 11

Introduction

Oracle's Berkeley DB 11gR2 release offers a perfect combination of technologies by including an SQL API that is fully compatible with SQLite. It is a great example of open source and commercial collaboration offering something for both camps. As a Berkeley DB developer, you now have a proven embedded SQL engine and C API to make application development even easier. And as an SQLite user, you have the option of a powerful, industry-supported storage engine that is built from the ground up for high-concurrency, and which includes powerful features like built-in replication and hot backup. The combination of the two technologies provides you with tremendous flexibility in that a single SQL API can now be used across a broad spectrum of applications ranging from the embedded space all the way up to large-scale transaction processing.

For the most part, if you have written code using SQLite, it should be easy to put Berkeley DB's SQL API to use. However, as with all things, there are a few technical caveats you should be mindful of. This paper explores the benefits and potential issues for those familiar with SQLite wanting to test drive the new Berkeley DB.

The Berkeley DB Architecture

The first thing to consider is — what the Berkeley DB (BDB) SQL interface is and what it is not. If you have worked with MySQL, you may be tempted to think of this new feature as an alternate backend storage module for SQLite, like InnoDB is to MySQL. It is not. Oracle has created an alternate front end for BDB — an SQLite compatible SQL API. So rather than SQLite having an alternate backend, BDB has an alternate front end. BDB is the frame of reference here.

Oracle has taken the SQLite source code, extracted everything from its storage layer up and grafted it on BDB. You could argue that this is simply semantics, however the combination of these two libraries presents some unique features and capabilities. For the sake of illustration, it is as if Oracle took the SQLite source code and completely replaced its pager and B-tree layers with BDB, as shown in Figure 1. "Berkeley DB Integration".

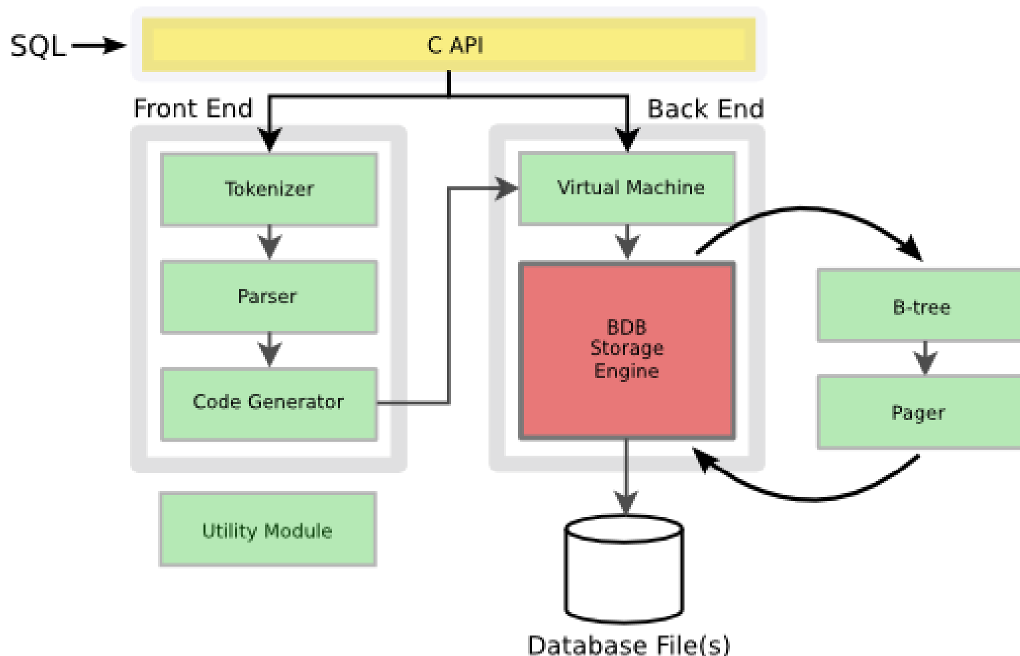

Figure 1. Berkeley DB Integration

As a result, though you may be working with the same SQL API up top, you are no longer dealing with the same database files down below. The representation of the database on disk is totally different. It is now a Berkeley DB database. As soon as you create a new database — say for example "foods.db", you have a totally different database file that SQLite knows nothing about.

You are now dealing with a Berkeley DB database and so all the standard Berkeley DB utilities, such as db_checkpoint, db_hotbackup, db_load, db_dump, and db_recover, can operate on it. While these utilities are for the most part optional, one such utility you definitely need to know about is "dbsql". This is the Berkeley DB equivalent of the "sqlite" command line utility, which operates on Berkeley DB created SQL databases.

Berkeley DB Benefits

If you are currently using SQLite, why consider switching to BDB? There are many operational differences between the two products that may impact your choice. One of the most important consideration is performance. There are many use cases where BDB provides significant performance improvements over SQLite. Latency and throughput are the most common measures in performance tests. Both of these measures are highly correlated with:

- Efficiency — the time taken for a single process to perform an operation.
- Concurrency — the number of concurrent operations possible per unit of time.

SQLite, by design, is engineered to be a "*portable, efficient SQL storage engine that offers maximum convenience, simplicity, in a small footprint*". Simple storage, rather than big-time concurrency, is the principal goal. The largest such design trade-off impacting performance is SQLite's use of database-level locking, which permits concurrent access to the database files during write operations, but there is a maximum of one writer who can be active at a time. As a result, SQLite's transaction rate stays more or less constant even when the number of concurrent connections (threads, processes, CPUs/cores) is increased. The performance of SQLite in concurrent write-intensive applications is limited to one thread.

Berkeley DB is built from the ground for operational efficiency while scaling. A critical component of that is concurrency, specifically large-scale write intensive concurrent transaction processing applications. Rather than using database-level locking Berkeley DB uses fine-grained (page level) locking. That enables BDB to handle multiple simultaneous operations working in a database at a given time (provided that they are working on separate pages). As a further optimization to increase concurrency, BDB supports what is known as "multi-version concurrency control" or "MVCC". Without MVCC a read operation blocks a write operation on the same database page. With MVCC the read operation maintains consistency by copying the page when the write requests the lock. In doing so, the write operation does not have to wait on the read transaction to commit its changes. Such optimizations allow for greater throughput and lower operational latency making Berkeley DB a much more concurrent solution for SQL storage.

Because of this, if you have an application that uses many concurrent connections to modify a database and page contention between the modifying threads is relatively low, then BDB can offer significant performance improvements, processing more transactions per second than SQLite for multiple writers. That said, for applications where data is rarely (or never) modified, SQLite and BDB have roughly the same locking overhead and so will perform roughly the same.

A more common use case is to have applications with a variety of create, read, update, and delete operations happening concurrently. In these cases, it is important to understand the potential for page contention to realize BDB's full performance advantage. The BDB database locking statistics (available via the dbsql ".stat" command or the db_stat command line tool) for page contention is an important consideration. If the majority of writers are working on the same pages at the same time then lock contention increases and throughput suffers. Essentially, if you had a BDB database where all of the writers were contending for the exactly the same page(s), then its performance is equivalent to SQLite because there is no advantage to page-level locking — everybody is accessing the same data at the same time so access must be serialized. Normally, as the size of the database file increases, so does the number of distinct pages within it and so the chance for page contention decreases. Therefore, the maximum transactions per second (TPS) scales with the number of concurrent connections (threads and/or processes).

As concurrency increases, the next most commonly encountered bottleneck is memory access and caching efficiency become the next limiting factors.

The following simple test scenario illustrates these performance issues. To start with, the number of writers is varied from 10 to 100 while observing the throughput of both SQLite and BDB. The transaction is made very simple consisting of a random UPDATE of that record, as follows:

```
BEGIN IMMEDIATE;
UPDATE test SET x=random() where id=ABS(random(10000));
COMMIT;
```

The intent of this transaction is to measure lock contention and overhead, so the simpler the SQL the better. The simplest UPDATE possible will do.

Furthermore, the test is designed such that each writer is theoretically working on a distinct page (and thus for BDB there is little or no page contention). To do this, a dummy table named "test" is created and filled with 10,000 records, where each record is roughly the size of a single BDB database page. Because each record (or row) requires its own database page the odds of two writers randomly selecting or updating the same record (and thus locking the same page) at the same time are very small (1 in 10,000).

The results of this test show that under these conditions, Berkeley DB consistently performs at a rate of about 8 times (800% faster) that of SQLite for 20 concurrent connections and greater, as shown in Figure 2. "Workload vs. Concurrent Connections".

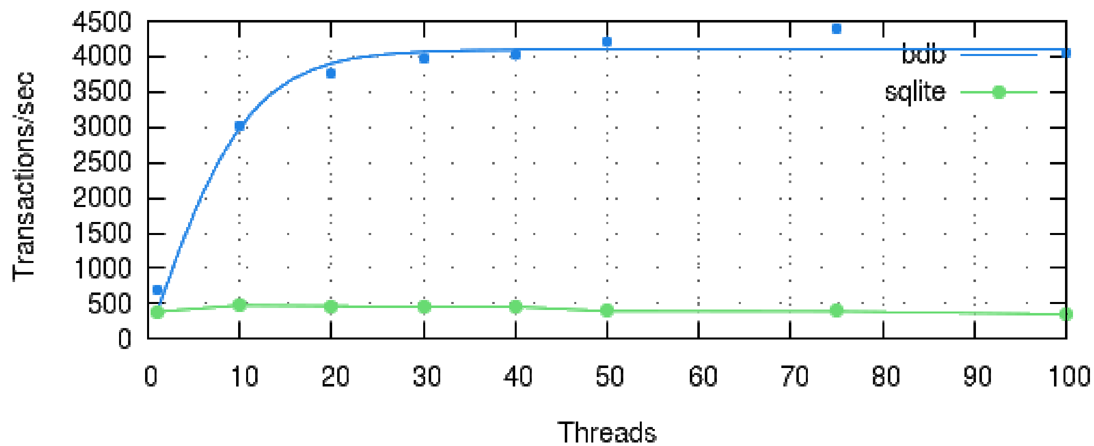

Figure 2. Workload vs. Concurrent Connections

The reason is that this is the optimal case in Berkeley DB. All writers are modifying distinct pages in the database because each row is on its own page – this essentially mimics row-level locking. While you see an eight-fold performance advantage here, recognize that this is a theoretical limit because a best-case scenario is created for this test case. In real life, this is not always going to be the case, as oftentimes applications tend to operate on a small subset of the data, forming "hot spots." Due to these contentious hot spots performance starts to fall from the optimal case to the degree to which writers contend for the same pages within them. If we assume that these hot spots comprise 60-70% of the update operations, you may then expect something more along the lines of 2-3 times the performance increase over SQLite on average for high-concurrency, write-intensive applications. Your mileage will vary, depending on data layout and data access patterns causing contention within your application. However, it is clear that BDB should provide better write scalability under concurrent load than SQLite in most cases. You should benchmark your application.

Berkeley DB Differences

Berkeley DB's SQL API is compatible with SQLite. Thus it enables it to act as a drop-in replacement for SQLite. That is, you add a single header file (db.h in addition to your existing sqlite3.h) and instead of linking the SQLite library you link against two Berkeley DB libraries: db-5 and db_sql-5. Re-compile, re-link, and you are in business. Your application that once used SQLite has been ported to use Berkeley DB.

That said, even though your code may be easily ported over to use Berkeley DB in place of SQLite, there are a few subtleties you should know about before deploying it into production applications. Though the API is the same across both systems, there are some significant operational differences. You need to realize that ultimately you are not using SQLite anymore. The API compatibility just refers to a communication protocol. And what you are communicating with is no longer the SQLite library, but the Berkeley DB library, which is totally different.

As a result, you need to make a slight adjustment in your perspective, factor in the Berkeley DB part and be mindful of both its similarities and differences. There are three areas that you need to have nodding acquaintance with:

- Configuration
- Contention handling
- Tuning

Configuration

As mentioned earlier, you are no longer working with the same database file(s) on disk. There are some significant differences here with SQLite. When you work with SQLite, you have a single database file (for example, foods.db) and during the course of open transactions a secondary journal file (for example, foods.db-journal). BDB uses a single database file (though in a completely different binary format than SQLite) and it uses what is called an "environment" directory. From the BDB SQL API documentation:

> In order to manage its resources (data, shared cache, locks, and transaction logs), Berkeley DB generally uses a directory that is called the Berkeley DB environment. As used with the BDB SQL interfaces, environments contain log files and the information that it requires to implement a shared cache and fine-grained locking.

To keep things somewhat similar, Berkeley DB uses the name of the SQLite journal file for the name of the environment directory. Thus, the BDB version of foods.db has an associated environment directory (not file) called foods.db-journal. And unlike the SQLite journal, which is transient, the BDB environment is a permanent, integral part of the database. Even when you back up your database, you should back up the associated environment – in fact it is best if you use the db_hotbackup utility (or better yet, review the Database and log file archival section in the Berkeley DB Programmer's Reference Guide). It is critical not only for configuration settings but also for database recovery. While there is much that could be said about the environment, perhaps the two most important things to mention are the log files and the optional database configuration file. The log files are simple — they hold a record of all committed transactions. In the event of a catastrophic failure (for example, system crash), the log files allow you to bring the database back to a consistent state. You can do this using the db_recover utility as follows:

```
$ db_recover -h <some-path-to-foods.db-journal>
```

You can alternately run db_recover from within the environment directory without any arguments and it will do the same thing (this assumes that the "foods.db" database file and "foods.db-journal" both exist in the same directory).

With each database, you have the option of creating a configuration file called DB_CONFIG in the environment directory, in the example that would be a file named "foods.db-journal/DB_CONFIG". You can then use this file to tweak certain settings and tune the database. There are many possible

settings, but you probably need only a handful of them at most. The following settings works well for heavy loads:

```
# Don't set these, use SQLite PRAGMA's
# set_flags DB_TXN_WRITE_NOSYNC
# set_cachesize 0 2147483648 1

mutex_set_max 1000000
set_tx_max 500000
set_lg_regionmax 524288
set_lg_bsize 4194304
set_lg_max 20971520
set_lk_max_locks 10000
set_lk_max_lockers 10000
set_lk_max_objects 10000
```

The set_lg_xxx parameters relate to log file settings while the set_lk_xxx parameters relate to the locking subsystem. The set_lg_bsize parameter helps with performance; it defines a memory region in which to hold log data. The larger this is, the more a transaction can run without having to flush write-ahead logging (WAL) data to disk. In the earlier tests this technique is used to adjust the lock settings by increasing them by a factor of ten, and the log buffer size by a factor of four.

Another parameter that proved to be important in heavy use is set_tx_max, which according to the documentation is defined as:

> Set the maximum number of active transactions that are supported by the environment. This value bounds the size of backing shared memory regions. Note that child transactions must be counted as active until their ultimate parent commits or aborts.

When 75 to 100 concurrent connections are run at full throttle, this number has to be raised considerably from its default. For most applications, the default values should suffice. But if you are putting the database under a heavy load, it is worth the time to read through the DB_CONFIG file Reference and learn how to tune the database appropriately.

Note: Some of these parameters only take effect by recreating the environment. To do this, you have to stop the application and run db_recover.

Contention Handling

An important and complex topic in SQLite is its locking model and initiating transactions in the right ways when dealing with multiple connections. This could be a separate article in itself, but here we will summarize the high level behavior. In SQLite, when you have multiple connections (processes, threads, and so on) operating on the same database, you need to pay attention to transaction semantics like BEGIN IMMEDIATE or BEGIN EXCLUSIVE depending on what you are doing. If you do not, you can end up with deadlocks.

In Berkeley DB, things are greatly simplified. All you need is just BEGIN in all cases. Berkeley DB automatically does deadlock detection for you. If your connection runs into a deadlock, Berkeley DB returns SQLITE_LOCKED. And then you just rerun your query. But with that simplicity, however, comes a tradeoff.

In many ways, Berkeley DB's mode of operation is very close to SQLite's "shared cache mode," which deviates significantly from the default transaction and locking semantics. But even then, it is still not an exact representation of SQLite's behavior. The main difference is how BDB handles database contention. Whereas SQLite by design favors asynchronous/non-blocking operations, Berkeley DB is just the opposite. What tends to be asynchronous and/or non-blocking in one may not be in the other. When you run a query in SQLite, you know right away whether you can run it or not; if the database is blocked, it returns "busy", informing you to try the operation again. Consider the following example:

```
sql = "update foods set name='JujyFruit' where name like 'Jujy%'";

sqlite3_prepare(db, sql, (int)strlen(sql), &stmt, &tail);

rc = sqlite3_step(stmt);

if(rc != SQLITE_DONE)

  {

     fprintf(stderr, "Error\n");

  }

sqlite3_finalize(stmt);
```

In SQLite, the return code "rc" may be SQLITE_BUSY, but the sqlite3_step() call does not block, ever. If there is no database-level lock serializing access, then it just works. If there is a lock, then you are told immediately and it is up to you what to do from that point.

In Berkeley DB, on the other hand, you never get SQLITE_BUSY: you do not get it back nor does your busy handler ever run (if you have registered one). There is no concept of busy in Berkeley DB (at least in its default mode of operation), because "busy" in SQLite signifies that the database is locked. Therefore, the sqlite3_step() method always runs to completion regardless, with the exception of a deadlock condition in which case you receive SQLITE_LOCKED.

In SQLite, even if the there is a lock in the way, the call still runs. The difference in BDB is that the call blocks. This is both good and bad. It is good because the call executes and you need not worry about

deadlocks as already pointed out. It is bad because your thread is committed to waiting however long it takes for the lock(s) to clear and the query to complete, whether you want to or not. It means that the option of doing something else is off the table: you are committed until the query goes through. So if you have threads or operations that vary according to priority, it is not possible to force the lesser to bow out. And it is possible for higher priority operations to have to wait for lesser priority and longer operations to run before they complete. That said, there is a way to change this behavior on a database at the database level using the DB_TXN_NOWAIT flag via the DB_CONFIG file with the following setting:

```
set_flags DB_TXN_NOWAIT 1
```

This causes the API calls not to block and return SQLITE_BUSY, replicating the exact behavior of SQLite.

Tuning

There is always a delicate balance between performance and durability. Applications can benefit greatly on either side of the continuum by understanding all the options. That said there are two important cache related configuration variables that you should know about when using either SQLite or Berkeley DB. They are the cache size and the synchronous setting.

Berkeley DB has tied existing SQLite PRAGMAs to analogous Berkeley DB settings, allowing you to configure a number of configuration parameters (normally set in the DB_CONFIG file) from within SQL. Two of the most important parameters are the cache size and synchronous settings, which can be set using the CACHE_SIZE and SYNCHRONOUS pragmas.

Caching in Berkeley DB is very similar to SQLite: it is a memory area used to cache recently read pages, as well as modified (dirty) pages that are used in a transaction. As a transaction modifies data, it fills the cache with the affected pages. When the transaction completes (commits), it writes the changes to the dirty pages out to the log file and later, during a checkpoint, it writes the dirty pages back to the database. If the cache is too small, it can fill up with dirty pages which then have to be evicted out to disk storage, which can be very slow. Therefore, having a cache that is sufficiently large to hold all modified pages and the commonly read pages (working set) offers optimal performance.

Note: As you can set the cache size in both the DB_CONFIG file and the PRAGMA, it is important to realize that the setting in DB_CONFIG is defined in bytes while the setting in the PRAGMA is defined in pages.

The synchronous setting flags control what happens to log records in the log buffer when a transaction commits. The log buffers are the first place database changes are committed in modifying a database and the primary bottleneck in that it involves disk I/O. There are two optional settings: DB_TXN_WRITE_NOSYNC and DB_TXN_NOSYNC.

The default settings are the most conservative and provide full durability (same setting used in these tests, DB_TXN_SYNC). Here, the log buffer is written to the log file and the log file is flushed with “fsync” or equivalent. If the operating system, file system, and I/O channel provide the guarantee they

are supposed to, then if there is a power failure or if the hardware crashes after the commit, the log records are preserved. When you recover the database (using db_recover), the committed transaction survives, and can be applied to the database bringing it back to a consistent state.

With the DB_TXN_WRITE_NOSYNC flag, Berkeley DB writes the log buffer to the log file when a transaction commits, but does not call "fsync". The log records then sit in a file system buffer, which persists if the program crashes but may not if the system crashes (as the OS may have cached part of the log files in memory during the time of the crash).

With the DB_TXN_NOSYNC flag, the log buffer is only written to the log file when it becomes full. Therefore, if the process crashes, any committed transactions that have not been flushed to the log file are rolled back when you recover the database.

Regardless of this setting, if Berkeley DB writes a dirty page from cache to a database file, it always ensures that all log records for that page are flushed to the log file (including the "fsync"). Therefore, write-ahead logging is maintained so that recovery always brings the database to a consistent state. So while these flags control how big the window of rolled back transactions can be, offering various performance characteristics, none of them permit corruption.

Conclusion

While this paper covers the major features and differences, this is at best just the tip of the iceberg of Berkeley DB as a whole. There is a lot more under the hood that SQLite users can learn about. Berkeley DB includes features like replication and hot backup. In the future, these features should start to become more integrated into the SQLite layer, perhaps through user-defined functions so that your application can more easily take advantage of them. But the first iteration is about just getting the API complete and working.

All in all, the combination of SQLite and Berkeley DB is a powerful one. It allows you to have a much wider range of applications using a single, open C API and SQL dialect. It is now possible to use this API for applications that range from large-scale transaction processing to tiny embedded environments that run within smart cards. You just pick the particular implementation that suits your needs.

You can download Oracle Berkeley DB at:

http://www.oracle.com/technetwork/database/berkeleydb/downloads/index.html

You can post your comments and questions at the Oracle Technology Network (OTN) forum for Oracle Berkeley DB at:

http://forums.oracle.com/forums/forum.jspa?forumID=271

For sales or support information, email to: berkeleydb-info_us@oracle.com

Find out about new product releases by sending an email to: bdb-join@oss.oracle.com

Berkeley DB SQL API vs.
SQLite API – Integration, Benefits and Differences
October 2010
Author: Mike Owens
Contributing Author: David Segleau

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment

Copyright © 2010, 2015 Oracle and/or its affiliates. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd. 0110