An Oracle White Paper

Oracle Berkeley DB SQL API vs. SQLite API – A Technical Evaluation

Executive Overview 1
Introduction 1
Overview 2
Methodology 2
Benchmarks 3
Results 4
Compatibility 7
Configuration 8
Behavior 9
Conclusion 14

Executive Overview

Oracle's Berkeley DB 11gR2 release includes an SQL API that is fully compatible with SQLite. Berkeley DB SQL API enables you to take advantage of the powerful features of Berkeley DB's enterprise grade transactional B-tree engine with the simplicity and ease of use of SQLite on the front end. The combination of the two technologies provides you with tremendous flexibility, in that, Berkeley DB SQL API can now be used to build applications across a broad spectrum ranging from the embedded space all the way up to large-scale transaction processing.

This paper demonstrates the potential for better performance by switching from SQLite to Berkeley DB's SQL API. Targeted test programs highlight the different performance characteristics between these functionally equivalent software libraries. This paper examines those tests, and presents a comparative analysis between Berkeley DB and SQLite. This paper also discusses the configuration and behavior differences between the two products that you must be aware of when migrating existing applications from using SQLite to Berkeley DB SQL API.

The results of the tests demonstrate that the Berkeley DB SQL API is not only very stable, but that its performance is superior to SQLite in write-intensive applications, under concurrent load and when stressed with heavy workloads.

Introduction

Berkeley DB's SQL API is compatible with SQLite and can act as a drop-in replacement for SQLite. For the purposes of this technical evaluation, the SQL API provided as part of the Berkeley DB 11gR2 (11.1.5.0.26) release is compared with SQLite (3.6.23), benchmarking the test programs on Linux and Solaris machines.

Overview

Hardware and operating system always impact the performance of a given application. For the purpose of this technical evaluation the focus is only on Linux and Solaris systems. Linux is used for general behavior tests while Solaris is used for performance tests. Solaris has superior timing and measurement tools, so the benchmarks were conducted using DTrace. The reasons for using DTrace are: it is an unparalleled tool in its ability to measure almost every aspect of process and system behavior and it makes taking such measurements (especially in multi-threaded environments) easy. Therefore, part of the project is written for and conducted within a Solaris environment. Performance characteristics are clearly different on Windows and the various embedded operating systems. However, the point of this paper is to illustrate the differences between these two products and not between operating systems.

Methodology

This section elaborates on the test methodology, tools, and platforms. You can download the test code from: http://www.oracle.com/technetwork/database/berkeleydb/learnmore/index.html

Linux

The generic behavior tests are designed to run on Linux. The first set of tests are illustrative of basic aspects of functionality. They are not comprehensive, but they do demonstrate the performance of the two products. The individual executables are built using Qtest's (Qt's library for testing[1]).

There are three main test files:

- **testbasics.cpp** — This contains simple, generic tests to open the database. It is basically a test of the test suite itself to ensure everything is built/linked and working.
- **testblocking.cpp** — This test illustrates one thread being forced to wait on another thread due to a page lock. For more information, see the Blocking section.
- **testdeadlock.cpp** — This test illustrates the response to database contention through response to SQLITE_ERROR. For more information, see the Behavior section.

[1] http://doc.trolltech.com/4.5/qtestlib-manual.html

Solaris

The Solaris test code is contained within the “tests/solaris” directory. To run the tests you must install some pre-requisite packages:

- GNU Compiler Collection (GCC)
- GNU Make
- Berkeley DB
- SQLite
- Ruby and Gnuplot (to plot the performance comparison graphs)

The Solaris tests use DTrace to gather timing information. For a non-root user, you must add the requisite privileges to enable the user to run DTrace. For example, to enable permissions for a user name “joe” as root, do the following:

```
bash $ usermod -K defaultpriv=basic,dtrace_proc,dtrace_user joe
```

Within the Solaris test’s project directory, you then run the “runtests.sh”, which compiles the source and DTrace files. Run all the tests and capture the raw data for all the graphs. Results of the tests are recorded in the “data.sql” file. To generate the performance plots, change to the “reporting” directory and run “make” which uses Gnuplot to chart the data collected from running the tests. The results are saved in the “tmp” directory.

Benchmarks

The goal for benchmarking code is to establish the relative performance of Berkeley DB’s SQL API and SQLite in high-concurrency, write-intensive applications.

There are a number of variables to consider in setting up the test conditions:

- **Cache Size** — The cache size is a critical setting. For the test workload, the default is too small and if you do not set sufficient cache size, the BDB library issues memory allocation errors ("Unable to allocate memory for transaction detail").
- **Page Size** — The test uses the default page size for both libraries. BDB's default is 16K, while SQLite's is 1K. Varying this setting in BDB can affect application performance[2].
- **Record Number** — More records result in more pages and more pages result in better overall concurrency. Hence, the example uses 10,000 records. Each record is approximately 400 bytes, so

[2] http://download.oracle.com/docs/cd/E17076_01/html/programmer_reference/general_am_conf.html#am_conf_pagesize

there are 40 records per page (16384/400) and therefore Berkeley DB uses about 250 pages to store this data. There may be a maximum of 100 threads running at any one time, hence there is less than a 50% chance that a given thread updates a locked page. Therefore, these conditions favor page-level locking over database-level locking.

- **Synchronous Setting** — The test uses the default setting for SYNCHRONOUS pragma for the following two reasons:
  - This is what both database engines consider to be fully durable transactions at the time of transaction commit.
  - It is difficult to accurately compare the alternate settings between the two databases as each setting does different things in each engine.

  Therefore, the default setting is the most practical assumption.

The test program is executed over a series of runs with the number of threads ranging from 1 to 100. It is important to note that the benchmarks reflect relative performance — how BDB performs relative to SQLite under the same conditions. The test environment does not attempt to represent optimal application coding options and so the results do not show optimal performance characteristics for any specific application; nor is it a representative of typical hardware — in this case the OpenSolaris instance is running within a virtual environment.

The test program consists of a multi-threaded application in which each thread performs a simple transaction consisting of a random select and a random update. You can download the core code from: http://www.oracle.com/technetwork/database/berkeleydb/learnmore/index.html

Results

The results are better than expected, clearly demonstrating a performance difference for concurrent workloads between the two products. As shown in Figure 1. "Workload vs. Concurrent Connections", Oracle Berkeley DB levels at about 4000 transactions per-second (TPS) scaling well up to 100 connections, whereas SQLite remains constant at 500 TPS, about one-tenth the performance of BDB. These numbers illustrate the fundamental differences between the different locking models used by the two products. SQLite's database-level locking constrains the system to a maximum of one write operation at a time even when using the write-ahead logging support in newer versions of SQLite (not tested in this paper). Thus, with SQLite there is a fixed maximum number of TPS regardless of the number of concurrent threads, available CPUs or cores for a given environment (operating system, hardware, and so on).

With Berkeley DB's page-level locking and data access pattern that minimize contention, it is theoretically possible that all BDB connections can write at the same time (concurrently). However, in this test case the hardware and the page level lock contention become the limiting factor. If you consider that a single BDB writer measures a throughput of about 700 TPS, then the theoretical limit would be 70,000 TPS with 100 non-conflicting concurrently executing threads. In these tests, on this hardware, lock contention is not the real limiting factor. It is more likely that other factors like I/O rates, the number of available CPUs, cores, and threads per-core (for hyper-threaded CPUs) are limiting BDB's overall throughput to about 4000 TPS.

Additional experimentation is possible by reducing the durability requirements of both solutions. When doing this, you can see dramatic changes in the performance characteristics of the test application. This is accomplished by turning "off" the SYNCHRONOUS pragma. Reducing the durability constraints reduces the processing overhead incurred by waiting on the I/O bus and disk latency. This can provide some insight into how much locking overhead is to blame when hitting the maximum TPS.

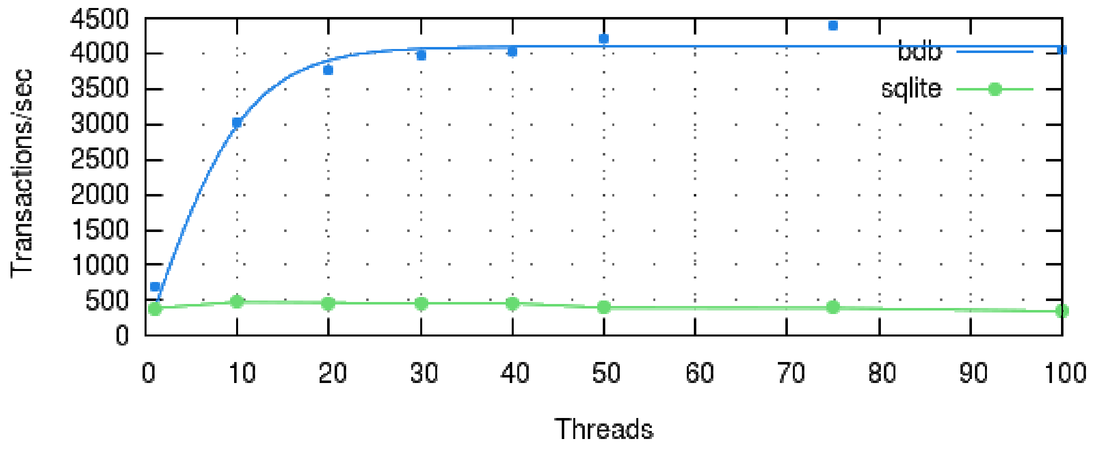

Figure 1. Workload vs. Concurrent Connections

Based on these measurements, BDB is more CPU intensive than SQLite, as shown in Figure 2. "CPU Utilization". However, when you consider that a 50% increase in CPU load results in an almost 300% increase in performance, it is a fair tradeoff.

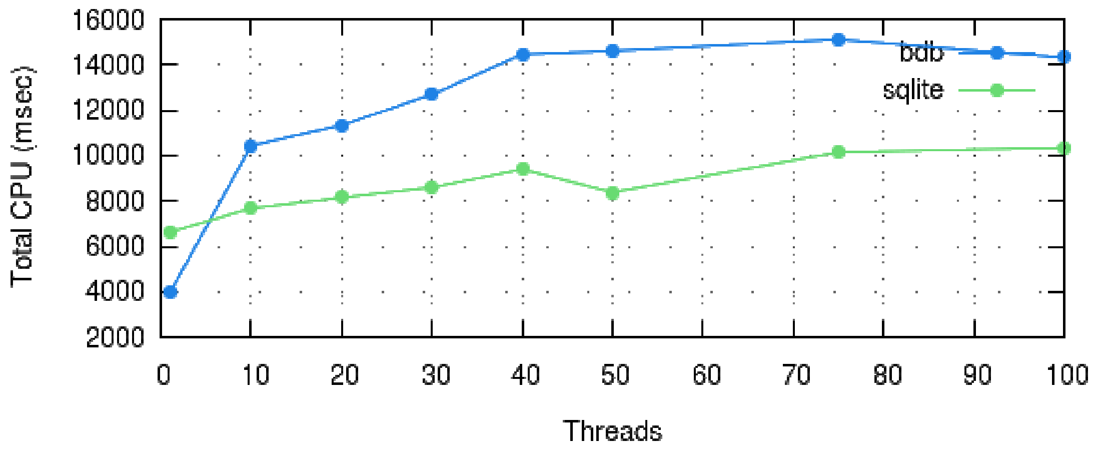

Figure 2. CPU Utilization

In addition, it was also observed that BDB spends less overall CPU time in system calls (Figure 3. "System Calls") as well as in transactions (Figure 4. "Transaction CPU Time"). Part of the reason for this is SQLite's busy-wait model (which requires multiple operating system calls) versus BDB's blocking model in dealing with lock contention. System calls require context switches between the user-mode application code and the protected kernel-mode operating system. In this case, BDB is actually performing more work, more efficiently and to greater effect than SQLite.

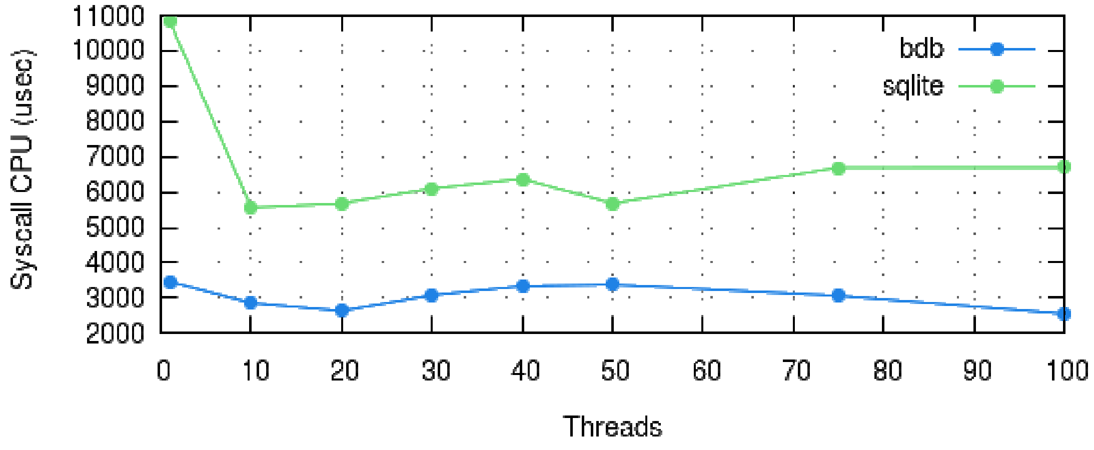

Figure 3. System Calls

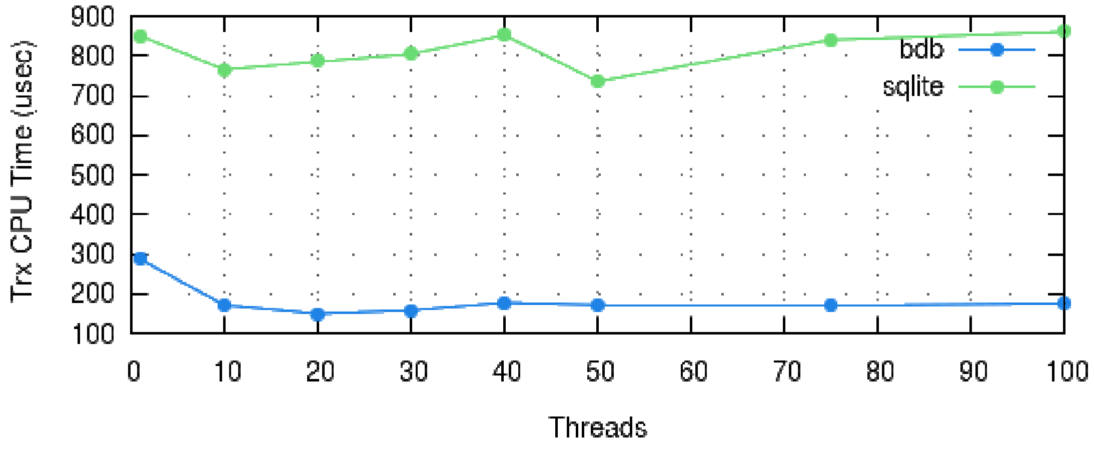

Figure 4. Transaction CPU Time

In SQLite, you have to busy-wait one way or the other, whether by handling SQLITE_BUSY or working with a busy handler. Either way it still amounts to a spinlock[3]. Spinlocks can make the situation worse by adding to overall resource consumption and they increase in number with concurrency (Figure 5. "SQLite Busy Handler Calls").

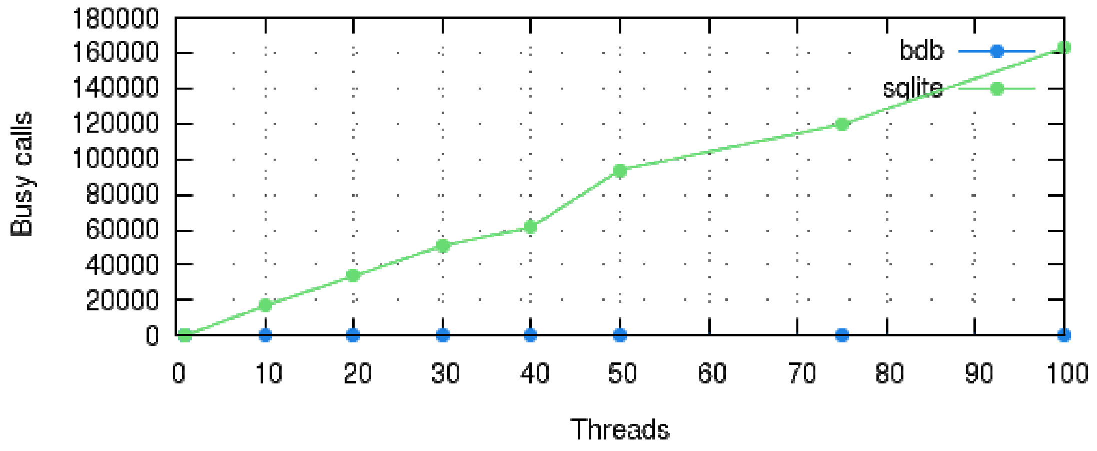

Figure 5. SQLite Busy Handler Calls

An SQLite application must wait, generally sleeping for some empirically determined amount of time, in the hope that when it retries the operation, the database is no longer busy. This is a tradeoff, in that, if the wait time is too long, performance is still suboptimal, but if it is too short then CPU cycles are still being eaten by the spinlock. Thus, SQLite uses more overall CPU time in system calls as well as in transactions due to spinlocks. All in all, the performance numbers sought for are there — BDB has very good throughput under a heavy load with many write operations, and that is the main objective of this test.

Compatibility

Berkeley DB's SQL API is compatible with SQLite's latest releases (version 3). Thus it enables BDB to act as a drop-in replacement for SQLite. As demonstrated by the performance tests, even though your code may be moved over from SQLite to Berkeley DB there are significant underlying implementation differences including a few subtleties you should know about before running it. This section explains some of the differences and issues involved.

[3] · http://en.wikipedia.org/wiki/Spinlock

Configuration

Both Berkeley DB and SQLite require more or less zero administration in deployment and require only the most basic of RDBMS systems administration skills during development. Database products are designed such that they always run well in a given configuration though they frequently run sub-optimally without some degree of careful configuration. Here, the focus is on the few configuration items to consider when using BDB. First and foremost is memory management. BDB allocates static memory regions to manage internal resources by mapping files into memory. To avoid memory allocation errors at runtime, sizing the memory regions properly when running many concurrent transactions or very large transactions, is important. The configuration variables that require attention in this case are the CACHE_SIZE pragma, which can be set from within the SQL environment, and the BDB-specific set_tx_max parameter, which must be set in the DB_CONFIG file.

The DB_CONFIG File

For large workloads Berkeley DB requires more resources than are provided in the default configuration. Configuring BDB-specific parameters can be done either using PRAGMA statements or via the DB_CONFIG file. Ensure you configure the DB_CONFIG file after running recovery and while the database is not in use. This provides the BDB storage layer with sufficient resources to support a large-scale application's database requirements. For the purposes of these tests, the following values are set in the DB_CONFIG file:

```
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
```

SQLite does not have anything similar to the external BDB configuration file (DB_CONFIG) so this is one administration difference worth studying before delving too deeply into BDB's SQL API. For more information on the DB_CONFIG parameters, see the Berkeley DB API Reference Guide that can be accessed from:
http://www.oracle.com/technetwork/database/berkeleydb/documentation/index.html.

Cache Size

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.

In these tests, given the dataset size and page size the CACHE_SIZE pragma should be set to at least 40,000 pages to avoid runtime page allocation errors. This is due to the highly concurrent nature of the test, as well as the row size and number of pages loaded in memory. Large or highly concurrent applications need to be aware of this configuration variable that must be addressed.

Behavior

There are two important migration issues that come to light when transitioning to Berkeley DB from SQLite:

- Deadlock Resolution
- Blocking Queries

BDB operates exclusively in a transaction model that is similar to shared cache mode in SQLite. This mode deviates significantly from the SQLite's default mode at both the API and SQL levels. Furthermore, the majority of programs that use SQLite are not prepared to deal with the semantics of shared cache mode.

SQLite's default transaction model facilitates writing programs that do not result in deadlock — they simply busy wait for the locked database. This may not be optimal, but it is conceptually easy to understand and to program too. For this technique to work, programs have to follow a very specific sequence of operations based on what they are doing (reading/writing) to avoid deadlocks. This technique hinges on SQLite's BEGIN TRANSACTION semantics. Without these semantics, deadlock detection in SQLite is impossible.

However this is not the case when using Berkeley DB's SQL API. In BDB's shared cache mode, SQLite's BEGIN TRANSACTION semantics do not work. All forms of BEGIN TRANSACTION are effectively reduced to the standard BEGIN, allowing all connections to plow right ahead. However, BDB does not fall into deadlocks like SQLite. Therefore, the approach to recognize and deal with contention is completely different and hinges on handling SQLITE_ERROR cautiously. If a program does not handle SQLITE_ERROR properly, then it results in a deadlock scenario which is undetectable to both BDB and the program.

Transaction Model

SQLite's concurrency model is different from most other databases. Some common patterns you use with other databases may not work like you expect in SQLite. To write good code, you must understand the transaction model, the locking model, cursors, and how they all relate to each other.

SQLite uses database-level locking, which is implemented using file locking on the database file. It keeps three different file locks to implement six lock states. To accommodate its locking model from the SQL level, SQLite has unique transaction semantics:

BEGIN	--> Read lock
BEGIN IMMEDIATE	--> "Modify" lock, called a RESERVED lock
BEGIN EXCLUSIVE	--> Write lock

There can be multiple read operations but only one write operation. The latter two are used as a kind of contract. If you can get them to complete (not get back SQLITE_BUSY), then you are guaranteed to be able to start modifying and eventually write to the database. Here modifying means, changing data by modifying pages in the page cache, which are not yet committed to disk (this would be writing pages which is done only in EXCLUSIVE).

There is a specific protocol you need to follow in SQLite to avoid deadlocks when writing applications that modify the contents of a database. Typically, it is best to start with the best transaction that supports the job at hand (reading or writing). If all you plan on doing is reading, then a simple BEGIN will do (which can be omitted whereby you run in autocommit mode). If you plan on modifying the database (INSERT, UPDATE, or DELETE), then start with BEGIN IMMEDIATE. By doing this, you hint that your application is about to enter a read-modify-write cycle and so SQLite waits for the correct locks in a deterministic way. By doing this, you know your application will not deadlock.

For example, here is a deadlock scenario when using SQLite. Say you have two concurrent programs X and Y connected to the same database, and neither knows anything about the other. X gets a READ lock. Y gets a RESERVED lock (READ and RESERVED can coexist — at this point you have potentially multiple read operations but just one modifier). X decides it wants to perform an UPDATE operation. So X tries, but is unsuccessful because Y has the only available RESERVED lock. Y decides it wants to commit its changes to disk and tries a COMMIT, which attempts to get an EXCLUSIVE lock. Y's attempt fails because it cannot get an EXCLUSIVE lock while there are any read locks on the database. Both X and Y just keep retrying in an endless loop until their operation succeeds — X's UPDATE and Y's COMMIT. They are now deadlocked. X has a READ lock and will never get the RESERVED lock because Y has it. Y has a RESERVED lock but will never get an EXCLUSIVE lock because it is blocked by X's READ lock.

The solution is simple: follow a simple protocol. Never start a transaction that may result in modified data with BEGIN. In our example, X should start with a BEGIN IMMEDIATE. By doing so, when it fails (because Y has an RESERVED lock), it falls back to an UNLOCKED state (does not hold a READ lock on the database) thus making it possible for any connection holding a RESERVED lock to obtain an EXCLUSIVE lock and complete. Thus when using this protocol, a RESERVED connection can busy-wait safely because it assumes that all other potential write operations start with a RESERVED lock and fall back into an unlocked state which does not interfere by introducing inhibitory READ locks. So everybody can safely busy-wait without fear of causing deadlock provided that they follow the proper BEGIN TRANSACTION semantics. Read operations always start with BEGIN and write operations with BEGIN IMMEDIATE. This protocol thus provides connections with means of getting out of each other's way.

The point is that SQLite's locking model provides ways to avoid deadlock, but you have to follow the pattern. Both BEGIN IMMEDIATE and BEGIN EXCLUSIVE are contracts that guarantee the state of the database wherein a write operation can assume that once it has either of these locks, it can proceed in a specific way (brute force) according to this protocol, and complete its work without encountering deadlock (assuming everyone else follows the proper protocol as well).

Contrast this with Berkeley DB's locking model. While BDB allows finer grain locking, its lock model changes these semantics, and so this protocol to avoid deadlock may in fact leave you in an unexpected application state. In BDB, X and Y would continue to execute even when they both use BEGIN IMMEDIATE. These two concurrent operations assume they must have been granted a RESERVED lock and can therefore proceed confidently to busy-wait, as they would with SQLite, until they are allowed to COMMIT. In fact, in BDB this is only true as long as they are not trying to modify the same **page**, something they cannot possibly (and arguably should not) know a priori. At this point, there are two new possible consistent, although unexpected, outcomes. Either or both operations could result in a SQLITE_LOCK if the BDB deadlock manager detects a deadlock, or they could block, wait for locks to be released and then complete their work. So, these unexpected behaviors could result from using a standard SQLite design protocol to avoid deadlock.

To see this, open two instances of dbsql and do the following in parallel, instruction by instruction in each session:

Session 1:

```
dbsql> create table a(x int);

dbsql> begin immediate;

dbsql> insert into a values (1);

dbsql> commit;
```

Session 2:

```
dbsql> create table b(x int);

dbsql> begin immediate;

dbsql> insert into b values (1);

dbsql> commit;
```

Both sessions operate concurrently after successfully starting a reserved transaction at the same time. Two read operations work simultaneously. This scenario works because they do not operate on the same page.

Now try this in both:

```
dbsql> begin immediate;

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

The second session is blocked. In this case, the block resolves once the first session COMMITs. But what would happen if the first session tries to update a page already locked by the second session — it leads to a deadlock.

Here is the scenario:

Session 1:

```
dbsql> begin immediate;

dbsql> insert into a values (3);

dbsql> insert into b values (3);

Error: database table is locked
```

Session 2:

```
dbsql> begin immediate;

dbsql> insert into b values (3);

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

Session 1 gets an error. But in the SQLite-based application code, this situation is entirely unaccounted for, as only those sessions that are in the shared cache mode would ever think to handle SQLITE_LOCKED. This scenario is the equivalent of a SQLITE_BUSY but without returning SQLITE_BUSY. So, the code is not prepared to handle this.

Going further, consider the following scenario (which is illustrated in the “debian/testdeadlock” unit test):

```
Session 1                                   Session 2


dbsql> begin immediate;                     dbsql> begin immediate;

dbsql> select * from a;                     dbsql> update b set x=1;

dbsql> update b set x=1;                    dbsql> update a set x=1;  (blocks)

Error: database table is locked

dbsql> commit;

tests/debian/test.db: previous transaction deadlock return not resolved

Error: SQL logic error or missing database
```

Session 1 locks a page on a. Session 2 locks the page on b. Session 2 then tries to update b and BDB causes it to block because the page is locked by Session 1. Session 1 then tries to update a, but gets SQLITE_ERROR (not SQLITE_LOCKED), which it is not prepared to handle. The only option available now is to abort the transaction and start over.

There is no deadlock, but the whole approach has changed. In the normal SQLite mode, you deal with contention by just trying the last SQL statement over. Now you have to deal with it by aborting the transaction and starting from scratch. Theoretically, all programs should be capable of dealing with SQLITE_ERROR. That is, handling SQLITE_ERROR (not SQLITE_BUSY or SQLITE_LOCKED) is an absolute requirement. But the problem with this also is that SQLITE_ERROR is vague. It is difficult to tell if the error is a result of contention or something more serious.

Even if Session 1 did receive and handle SQLITE_LOCKED, it still has to abort the transaction and start from scratch. This is because all further attempts lead to a potential deadlock, in which Session 2 is permanently blocked and keeps Session 1 from ever completing. So in this situation, it is not clear as to who has the right of way. The semantics change so that if you receive SQLITE_LOCKED or SQLITE_ERROR, the only safe option you have is to abort the transaction and start over. This is perhaps the unavoidable consequence of having page-level locking. But in any case, it is a significant change from the default mode of operation that SQLite uses, and you must examine and test the migrated code carefully.

Blocking

Blocking is another issue as it is a deviation from the expected behavior in the SQLite C API. Berkeley DB essentially has no support for SQLITE_BUSY and the associated busy handler callback because a BDB database is not "busy", meaning locked at the database level, ever. Thus, when the program issues a query, it may potentially block until that query can complete. While the SQLite API is uniformly asynchronous and has no method by which to block, the BDB version is just the opposite, having no method by which to not block. And while BDB's deadlock detection does not cause blocking (returning an error), page locks that are not detected as deadlocks induce blocking.

The problem is that a connection that does not need to perform a given SQL operation — should the database be busy — is nonetheless fully committed the moment it issues a query. Furthermore, it is then totally dependent on the connection that has the opposing lock and is not able to be released until its transaction completes. This makes it possible for higher-priority threads to be held captive by lower priority operations. This is illustrated in the "debian/testblocking" unit test.

Note that the BDB SQL API's long-term goal is to provide the value of a more concurrent transactional solution to SQLite programmers, with the absolute minimum required changes. This means that as Berkeley DB's SQL API matures it is possible that these issues will go away, thus allowing any SQLite program to transition and know that both the explicit contracts in the ANSI C API as well as the implicit contracts of best practices, design patterns, protocols, and so on, all work as expected with some additional benefits. The goal is to be 100% compatible, and that goal extends beyond the API into the behavioral aspects.

Conclusion

Overall, tests and subsequent analysis show that BDB SQL API is very stable and clearly BDB's performance and concurrency is superior to SQLite in write-intensive applications and heavy workloads, allowing it to get higher TPS throughput. Furthermore, Berkeley DB uses fewer system calls and spends less time inside of transactions for these kinds of applications.

SQLite applications that opt to use the BDB SQL API should have no problems with the transition. Although BDB SQL API is compatible with SQLite and can be treated as a drop-in replacement for SQLite, you must be aware that it is not 100% identical with SQLite, due to some of the behavioral and configuration differences described in this paper. As a developer of existing applications, you need to be mindful of these migration issues to take advantage of BDB's features and benefits.

Berkeley DB SQL API vs.
SQLite API – A Technical Evaluation
November 2010
Author: Mike Owens
Contributing Authors: Greg Burd and 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