Testing Berkeley DB

Ashok Joshi, Charles Lamb, Carol Sandstrom
Oracle Corporation
{ashok.joshi,charles.lamb,carol.sandstrom}@oracle.com

Abstract

Oracle Berkeley DB is a family of database engines that provide high performance, transactional data management on a wide variety of platforms. Berkeley DB products are available under a dual license: an open source license and a commercial license. We discuss some of the standard testing and tuning techniques used for ensuring the quality and reliability of the Berkeley DB library, emphasizing some of the interesting testing challenges arising due to multi-platform support. Since Berkeley DB is available in source code form, it can be adapted/modified by users in the field. It is necessary to test and validate the modified version of Berkeley DB before it can be deployed in production. We discuss some testing tools and techniques provided with the Berkeley DB distribution that simplify the process of user-testing and certifying Berkeley DB ports to new platforms.

1 Introduction

Database software is complex along many dimensions: large number of features and APIs, concurrent read and write activity, fault-tolerance and recovery, performance, scalability, and reliability. In a wide variety of situations, database applications manage mission critical data, and there is an implicit assumption that the underlying data management services are well tested, reliable and correct. This article discusses some of the testing and tuning methodologies and practices used to ensure high quality for Berkeley DB, a family of embeddable database engines.

Oracle Berkeley DB [1] is a family of database engines that provide robust data management services in a wide variety of usage scenarios ranging from enterprise-class applications to applications running on mobile devices such as cell phones. Berkeley DB products are distributed under a dual license - the open source GPL-like license for open source applications, and a commercial license for closed source applications.

It is important to highlight some of the differences between an open source product such as Berkeley DB, and proprietary, closed source products. Note that Berkeley DB is not an *open development project*; Berkeley DB products are developed by a dedicated group of software engineers. Berkeley DB products are distributed in source code form, complete with a comprehensive test suite. Source code distribution adds an interesting set of development and testing challenges, since the user is free to choose from a variety of compilers and development environments to build Berkeley DB and the application. Further, a small number of users can and do modify the Berkeley DB sources, primarily for porting to new platforms; the Berkeley DB distribution includes a test suite that can be run by end users in order to validate their changes.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

The rest of this paper is organized as follows. We begin with a description of each of the products. This is followed by a discussion of some of the *static testing* tools that we use internally to verify the correctness of the code. This is followed by a section on testing - this includes unit testing, stress testing, performance testing and analysis as well as ad-hoc, use-case-specific testing. Next, we discuss the Berkeley DB approach to portability, platform support and testing. Portability is particularly interesting in the Berkeley DB context because we allow and encourage our users to port Berkeley DB to the platform of their choice. The Berkeley DB distribution includes a platform test suite designed to exercise the platform-specific aspects of the port.

Oracle Berkeley DB has benefited tremendously from a large and active community of users who test the products, review the code, report problems and suggest enhancements and features. The involvement of the user community has been critical to the success of the Berkeley DB products.

2 Berkeley DB product family overview

The Berkeley DB product family consists of three products: Berkeley DB, Berkeley DB Java Edition and Berkeley DB XML. Berkeley DB products are available as libraries with simple, proprietary APIs for data access and manipulation as well as database administration. Berkeley DB does not support SQL, though it has been used as the storage engine for SQL database products. A typical Berkeley DB application makes API calls to start and end transactions, store and retrieve data as well as to perform administrative functions such as checkpoints and backups. Thus, a Berkeley DB application is completely ”self-contained” with respect to all data management activities; this enables a *zero manual administration* approach to application development. This capability is critical in a large number of applications including embedded applications, where manual administration is impossible.

Berkeley DB Java Edition is a 100% pure Java implementation whereas Berkeley DB is implemented in C; both products have similar APIs and capabilities for data management. Berkeley DB XML (implemented in C++) is an XML database engine with XQuery and XPath capabilities; Berkeley DB XML layers on top of Berkeley DB and uses it for storage, indexing, transactions and other database capabilities.

Although Berkeley DB and Berkeley DB Java Edition are very similar with respect to the features and functionality they provide, architecturally, they are quite different. From a testing point of view, porting is not as big an issue for Berkeley DB Java Edition, since the JVM is inherently portable. Both products provide indexed access to data; Berkeley DB supports B-trees as well as hash indexing whereas Berkeley DB Java Edition only supports B-tree indices. Both products support concurrent access to data. Berkeley DB permits concurrent threads, or concurrent processes or both, whereas Berkeley DB Java Edition typically supports multiple threads within a process and more limited multi-process access. Both products support transactions, including support for the various ANSI isolation levels. A row is simply an opaque *key:value* pair; Berkeley DB does not have the notion of data types, but the Direct Persistence Layer of Berkeley DB Java Edition does provide an optional schema-like capability. Interpreting the opaque contents of the row is left entirely up to the application. APIs for administrative operations like database checkpoints and backups are provided by all three products.

Architecturally, Berkeley DB is similar to the ”update-in-place” architecture of most other traditional database systems. Berkeley DB Java Edition, on the other hand, uses log-structured storage for managing on-disk data. Every change results in a new entry in the log. A separate garbage collector thread that runs in the background reclaims space occupied by obsolete data. Thus, though there are commonalities between the test suites for Berkeley DB and Berkeley DB Java Edition with respect to testing API behavior, the Berkeley DB Java Edition test suite also contains specific tests for exercising the log cleaner, ”out of disk space” scenarios, log archiving and other aspects specific to the log structured architecture of Berkeley DB Java Edition.

Berkeley DB XML, on the other hand, manages XML documents. Documents can either be stored as whole documents, or as individual nodes. Berkeley DB XML creates indices on various attributes to improve access performance. Since Berkeley DB XML is layered on the Berkeley DB database engine, it can leverage the test suite of the underlying storage engine, including the replication and high availability features. Berkeley DB XML also leverages the XML standards specifications in order to test the correctness of XML processing.

2.1 Feature sets

Early on in the history of Berkeley DB development, we made the decision to provide a variety of feature sets for the Berkeley DB products. Applications that use Berkeley DB have varying data management needs; rather than a ”one size fits all” approach, Berkeley DB products offer the user a choice of which features to use. Further, an application using the simpler feature sets can build a small footprint Berkeley DB library; this is particularly important for applications running on resource-constrained devices such as mobile devices.

The simplest feature set is called *Data Store* (available in Berkeley DB and Berkeley DB XML). This allows either a single writer or concurrent readers. The data store option is ideal for simple applications that need high performance indexed access to data without the need for read-write concurrency or transactions.

The *Concurrent Data Store* feature set allows concurrent readers and one writer, but without transactions and recovery. This is most suitable for situations where simplicity, footprint and performance of the application are more important than data consistency or integrity. In most of these situations, the data managed by Berkeley DB is either transient data, or the data can be retrieved from another (perhaps transactionally managed) data source in case of data loss.

The *Transactional Data Store* feature set provides the full set of features including concurrency, transactions, logging and recovery. This is the option of choice for applications that have stringent data consistency and integrity requirements. As mentioned earlier, Berkeley DB also provides APIs (and standalone utilities) for administrative functions such as backups, checkpoints and recovery, further simplifying the task of building zero manual administration applications.

Finally, Berkeley DB provides the *High Availability* (HA) option for applications that need multi-node scalability and extremely high availability. Berkeley DB HA supports a single read-write master and multiple reader configuration implemented via log shipping. If the master fails, a new master is elected and processing continues uninterrupted. Berkeley DB HA also supports a variety of options for improving transaction performance; for example, it is possible to commit a transaction either by writing the commit log record to the master’s local disk *(commit to disk)*, or by sending reliable commit messages to a majority of the secondaries *(commit to the network)*.

The Berkeley DB design philosophy has always been to provide mechanisms, not policy. This provides tremendous flexibility to the application developer with respect to choosing the features to use as well as configuring memory, IO, network traffic, disk usage and other system resources. This level of flexibility implies multiple permutations of choices and configurations during testing and tuning.

3 Ensuring product quality

In general, we follow the *extreme programming* methodology for unit testing - implement the test first, then implement the code. This methodology results in much better code quality in the minimum amount of time. We’ll discuss unit test development in more detail below.

We use a combination of code reviews and software tools in order to ensure the correctness of the new implementation. A good code review is highly effective in ensuring better code quality. Tools such as *lint* can detect potential problems such as uninitialized variables, type-incompatible assignments and incorrect arguments. Tools such as *Purify* [3] check for memory leaks and potential out-of-bounds references. Our development methodology requires that code is peer-reviewed and checked for memory leaks, adherence to language and portability standards and standard coding conventions. All *lint* inconsistencies are fixed before the code changes are approved. Since Berkeley DB products are distributed in source form, we periodically compile and build the product using a wide variety of compilers and address compiler issues and warnings.

3.1 Static testing

The term *static testing* refers to various tools for measuring code coverage, memory corruption and memory leak checkers such as *Purify*, and programs such as *lint* that are used to identify problems in the code and/or tests.

Code coverage is a very useful technique for determining the effectiveness of the test suite. In a code coverage run, the code is instrumented to monitor coverage, and then the entire test suite is run to determine which code blocks are executed, and which code blocks are not exercised. Code coverage testing is done periodically since analyzing the results and adding new tests can be a significant amount of work.

A code coverage run will highlight the lines of code that are not exercised. Rather than a ”brute force” approach to achieving very high code coverage, we focus on developing tests that target the important code paths. Code coverage results need to be interpreted carefully since they do not indicate whether every possible code *path* was exercised. For example, if there are multiple ways of reaching a particular code block, a code coverage run will identify that code block as covered even if the test only exercises one of the ways of reaching that code block. It may be necessary to use other techniques such as logging and/or using a debugger to ensure that tests exercise specific code paths. This certainly improves the quality of the tests, but may not result in increasing code coverage.

We run *Purify* periodically to identify and eliminate memory leaks. We re-compile, build and run tests periodically on many different families of systems including Linux, Windows, Solaris, HP-UX, AIX, and others. Testing starts with the compiler native to a system, but we also test non-native compilers in situations where they are commonly used, e.g. *gcc*. We also make a point of testing different versions of the same compiler. Build failures are fixed when possible and silenced when necessary to avoid the possibility of real failures vanishing in the noise of unimportant warnings. All engineers receive e-mail from automated *lint* runs daily and are responsible for fixing errors found in their code.

3.2 Regression Testing

Test execution is automated. Under normal circumstances, the test suite is run on two or three different platforms concurrently. Builds are verified at least daily. Tests generate detailed logs of the execution, and the scripts supporting the tests automatically report build and test failures by email. The QA group analyzes the results of each test run; test failures are fixed by the QA group, whereas programmatic errors are reported to the relevant developer. Code changes are logged in CVS with tracking numbers so it is usually possible for a QA engineer to pinpoint the piece of code that changed and inform the appropriate development engineer in order to address the issue expeditiously.

3.3 Unit testing

We follow the extreme programming principle of ”test first, code later” in developing unit tests. We first implement a set of tests that are designed to verify the correctness of the change. Depending on the scope of the change, developing unit tests can be a significant effort. The developer responsible for a certain feature usually develops the required unit tests, with input from other team members. The QA group expands and standardizes tests from development. Having the unit tests ready before the feature is developed has several benefits including potential improvement of the design and eliminating the possibility that the feature will go untested.

The complete unit test suite is included with each Berkeley DB distribution and can be run by the end user who needs to verify their specific Berkeley DB application implementation and deployment.

3.4 Instrumenting the code for testing

Sometimes, it is necessary to add special code to simulate certain conditions for testing (e.g. IO failure). In such cases, we instrument the code with assertions and hooks. For example, we use Java *assertions*; assertions are enabled only during debugging and testing. We also have *assert* statements that allow us to force *IOExceptions* to be thrown at specific points in the code and these points can be specified by the test program. These simulated write failures ensure that the error handling code is correct.

3.5 System and Stress testing

System and stress testing is designed to test the end-to-end behavior of the software. We use a parameterized driver program; this enables us to easily tailor a particular test run to exercise specific aspects, features and configurations. For example, it is possible to select the number of threads, the ratio of writes to reads, memory size and various configuration parameters.

The driver is normally run in randomized mode to force the testing of new combinations, and is routinely run on multi-processor machines (even slow multi-processors) to increase contention. The driver program runs for extended periods of time and exercises the various Berkeley DB APIs with the objective of finding a problem. Since the driver program is well parameterized, it is possible to use the same program not only to stress test specific aspects of the software but also to measure performance of basic operations.

Running system and stress tests is an on-going activity. Problems identified by unit tests are usually reproducible and hence relatively easy to analyze. On the other hand, diagnosing and fixing problems found by system and stress tests is harder since it is not easy to reproduce the problem. Stress tests run for extended periods of time, making it difficult to reproduce the exact state of the system at the time of the failure. Berkeley DBs extensive logging capabilities are extremely helpful in analyzing system test failures.

Testing Berkeley DB HA requires a test harness that can exercise a distributed application running on multiple nodes. We are in the process of developing a test harness based on *Erlang* [2].

Developing comprehensive system and stress tests is always a challenging task, and an on-going process. Recently, we encountered a customer issue which highlighted a limitation in one of our stress tests that exercises the *multi-version concurrency control* feature in Berkeley DB. Multi-version concurrency control requires additional memory in the buffer pool in order to store previous versions (snapshots) of database pages; when there is no more room in the buffer pool, Berkeley DB temporarily overflows the snapshots to disk. Though overflow to disk for snapshots is supported, the expectation is that the user will configure the buffer pool so that overflowing to disk is rare. In this particular customer situation, a combination of a long-running writer transaction, multiple reader transactions and a small buffer pool resulted in a large number of snapshots being written to disk. Further, there was a bug in the ”read snapshot from disk” code, which resulted in the incorrect version being returned to the transaction.

It took a significant amount of investigation to recreate the scenario and diagnose the problem, since the problem was not easy to reproduce. Fortunately, once the problem was identified, the fix was very easy (just a few lines of changed code). Needless to say, we have added stress tests to exercise the ”overflow snapshot to disk” scenario.

3.6 Performance testing and analysis

Performance testing and analysis is an on-going activity. As mentioned earlier, the system test driver program is parameterized; this enables us to use it to measure the performance of various operations.

We maintain performance data history for all releases in order to detect regressions. This is particularly helpful during the development of new features and functionality, since we can quickly identify and fix performance problems that are introduced by the new code. Our experience suggests that measuring the performance of basic operations is sufficient to identify performance regressions in most situations; a complex test is not required.

We often get requests for performance data for certain, customer-specific workloads. The customer workload usually has specific requirements for record size, number of records in the database, throughput and response time constraints and so on. Having a simple, parameterized driver program is tremendously helpful in being able to respond quickly to such requests. In most cases, it is possible to easily modify the driver program in order to approximate the specific workload and generate performance data.

3.7 Release testing

In addition to the continual testing during the development phase, we run an additional set of tests after a release is code-complete, in order to verify some of the uncommon platform configurations and to ensure that add-on modules (like *Perl* - *http://perl.com*) are tested and ready for release.

Berkeley DB has been in widespread use for more than a decade and use of historical versions is quite common. Release testing also includes upgrade tests, to verify that databases from earlier versions can be seamlessly upgraded to the new version.

3.8 Platform porting and testing

Berkeley DB products are different from most commercially available database systems because Berkeley DB is shipped in source code form (along with the source code for the tests). This makes it convenient for our users to port Berkeley DB to the platform of their choice. We often work jointly with our customers on such porting efforts.

Portability is not an issue for Berkeley DB Java Edition; the rest of this discussion applies mainly to Berkeley DB and Berkeley DB XML. By design, Berkeley DB products adhere strictly to programming language standards and have minimal dependence on platform primitives. As is the case with other portable software products, Berkeley DB isolates the operating system dependent code to a small set of code modules. This ensures that port-specific differences are localized.

Berkeley DB supports a long list of popular platforms. Each new version is released simultaneously on all supported platforms. This is achieved by continuous and frequent testing on a wide variety of platforms in a round-robin manner. Though it is not very common to find platform-specific problems, the advantage of this approach is that such problems are identified early in the development stage.

Occasionally, a customer requires Berkeley DB on a platform that is not already supported. In order to assist our users in porting Berkeley DB, we have developed a porting guide and a platform-specific test suite in C. Though not as comprehensive as the full test suite, this compact, but complete test suite is designed to thoroughly exercise the various operating system primitives that Berkeley DB uses. The tests can also be modified to suit the requirements of the underlying platform. This is especially critical when testing on resource-constrained devices such as mobile phones.

A typical customer-porting scenario is as follows. The customer will download and build the source code on the target platform. This can be an iterative edit-and-build process. After Berkeley DB is built successfully, the user can run either the full test suite (if the platform is capable) or just the platform-specific test suite. When all tests execute successfully, the user can be confident that Berkeley DB will run on the target platform.

If the user had to make changes to the code, build scripts or tests, we request them to send us the changes so that we can incorporate them into future releases.

We recently had a very positive experience where a customer worked with one of our field engineers in Japan to demonstrate that Berkeley DB could be ported to a new platform easily and painlessly. They used the porting guide and tools provided with the Berkeley DB distribution in order to compile, build and validate Berkeley DB on the new platform in less than two months. Typically, a port to a new platform of a commercial database products takes many person-months of work, so this is a remarkable achievement.

4 Tuning

The Berkeley DB philosophy is to provide mechanisms, not policy. Berkeley DB (like other DBMSs) provides a large number of ”knobs” to influence the run-time behavior and performance of the system. The user can control system parameters such as amount of memory, threads, synchronous vs. buffered IO etc. The user can also choose to enable or disable features such as transactions, locking and multi-version concurrency control.

Choosing the various parameters appropriately requires a good understanding of the system; this is further complicated because some choices have dependencies on other choices and settings.

Berkeley DB has a comprehensive statistics and logging facility that provides useful data to aid tuning. Berkeley DB documentation provides detailed information on the various parameters and settings available to the user. Further, there are several source code sample programs included with the distribution that illustrate how certain parameters may be used. The Berkeley DB discussion forums are an excellent source for getting advice and feedback on tuning Berkeley DB. In specific situations, we provide customer-specific consulting for performance analysis and tuning. Finally, having access to the Berkeley DB source code can be helpful in understanding and tuning the software. On a number of occasions, users have been able to achieve significant (ten-fold or more) improvements in performance by modifying just a few Berkeley DB parameters.

We are planning to develop a utility that will interpret the statistics and make recommendations. We are also considering integration with other comprehensive monitoring and tuning utilities such as Oracle Enterprise Manager.

5 Conclusions

Exhaustive testing is fundamental to the quality and success of the Berkeley DB family of products. We pay attention to testing, code quality and performance throughout the development cycle. In terms of lines of code, the test suite is about 40% of the lines of code in the products and it continues to evolve along with the products.

Acknowledgements

We work closely with our user community in order to improve the products as well as to port to specific platforms. Berkeley DB products have benefited tremendously from user feedback and help. A large portion of the credit goes to the excellent development team; their expertise and painstaking attention to quality, performance and overall maintainability of the code continues to be instrumental in delivering world-class products.

References

[1] Berkeley DB Documentation: `http://www.oracle.com/technology/documentation/berkeley-db/db/`

[2] Erlang: `www.erlang.org`

[3] Purify: `www.ibm.com/software/awdtools/purify`