An Object Oriented Framework For Database Benchmarking Based On The Open Source Database Benchmark Project

Jason S. Vorpahl and Paul J. Wagner
Department of Computer Science
University of Wisconsin Eau Claire
Eau Claire, Wisconsin 54701
vorpahjs@uwec.edu wagnerpj@uwec.edu

Abstract

This paper describes two research contributions. First, we have completed a prototype implementation of the Open Source Database Benchmark (OSDB) project for the Oracle database management system. Second, we have developed an implementation of an extensible framework for database benchmark tools based on the OSDB project. The framework allows developers to easily "plug-in" database systems and tests which would allow for a complete database benchmarking suite. To achieve a balance between gaining flexibility and reducing overhead, we use an object oriented design that incorporates design patterns to solve several significant problems seen with the current system.

1. Introduction

In recent years, database management systems (DBMSs) have become a central back-bone to many computer systems and applications where data throughput has become top priority. Financial giants such as the Yahoo! Corporation and Continental Airlines Inc. are running dozens of DBMSs that help maintain everything from web searches to airline tickets [7]. Since throughput is such an important issue in the storing and retrieval of data, there needs to be a scientific way to compare the throughput of DBMSs against each other on the same operating systems, and compare the same DBMS's throughput on different operating systems. A benchmark is one scientific way to achieve such results.

The Open Source Database Benchmark project (OSDB) grew out of the need of Compaq Computer Corporation to evaluate the I/O throughput and general processing power of GNU Linux/Alpha. The benchmark test suite is based on AS3AP, the ANSI SQL Scalable and Portable Benchmark, which was documented in Chapter 5 of "The Benchmark Handbook", edited by Jim Gray [3]. The AS3AP test specification was developed by D. Bitton and C. Turbyfill.

Although, OSDB is complete in its implementation, one problem that has plagued the project is the complexity to implement a wide variety of DBMSs due to the different ways each DBMS's C API is developed. This was discovered by past research projects that attempted to implement an Oracle DBMS port for OSDB. Since OSDB is implemented in the C programming language and naturally thus not object oriented, we began running into coupling problems with the code while further developing an Oracle 10g port.

The purpose of this paper is to present an object oriented framework that makes it as easy as possible for developers to add new databases to the testing suite, but still re-use existing tests. We used many of the design patterns from the classic "Gang of Four" Design Pattern book and based our framework for running the individual tests from the open source JUnit project.

2. Background

2.1. ANSI SQL Scalable and Portable Benchmark and OSDB

The ANSI SQL Scalable and Portable Benchmark for relational database systems is a non-commercial benchmark that came about from the need for a general benchmark that can be used to compare relational database systems with vastly different architectures and capabilities over a variety of workloads. AS3AP is designed to provide a comprehensive but tractable set of tests for database processing power, have built-in scalability and portability so that it can be used to test a broad range of systems, minimize human effort

in implementing and running the benchmark tests, and provide a uniform metric for a straightforward and non-ambiguous interpretation of the benchmark results [3].

There are several differences between AS3AP and OSDB. AS3AP requires a complete SQL implementation in order to run. OSDB is designed to accommodate incomplete SQL implementations, and even non-SQL implementations. AS3AP also includes a test to do journal recovery. OSDB omits this type of test because of the rare event of database recovery. It is of minor importance to measure a database recovery when analyzing the day-to-day performance of a system. Other changes, such as test grouping and ordering, reflect an attempt to isolate one-time events like initial database creation from day-to-day operations [6].

2.2. Previous OSDB Implementation

The current release of OSDB as of October 19, 2004 is version 0.17. This release is implemented in the C programming language and uses the M4 macro language to provide decoupling between the test cases, database implementations, and the program driver. The application is built using GNU Autoconf and Automake to automatically configure the source code package and generate make files compliant with the GNU coding standards. Version 0.17 of OSDB provides ports for Informix, PostgreSQL, and MySQL DBMSs, and a prototype port the commercial DBMS Oracle that was a portion of a previous research project by Dr. Paul J. Wagner and student Justin Sabelko from the University of Wisconsin – Eau Claire Department of Computer Science.

3. Work Done/Contributions

Since October of 2004, we have completed two major accomplishments pertaining to OSDB. We have implemented and tuned the single user port for Oracle 10g and developed a redesign of the benchmark framework based on OSDB and JUnit.

3.1. OSDB Oracle 10g Port

We inherited the Oracle port prototype from a previous research project and determined that there was still additional fine tuning that needed to be done in order to consider the prototype complete. There were inconsistencies dealing with creating cluster and hash indexes, opening and closing the cursor, and fetching the cursor. There were also some performance issues with the SQL Loader utility that is used for moving data from external files into the Oracle database. While attempting to fix these problems, we hoped to learn some of the problems that the current implementation of OSDB has, and come up with good ideas on how we could attempt to fix them through a redesign of the application.

We found that Oracle 10g creates B-tree indexes by default but that the AS3AP database benchmark test assumes that three types of indexes are supported. Those indexes are a B-Tree clustered index, a B-tree secondary non-clustered index, and a hashed secondary index. We spent a bit of time attempting to find a way to create clustered indexes and hash indexes within the context of the application but found this next to impossible because Oracle requires some vendor specific syntax during table and index creation. This was in contrast to the DBMSs that the software was originally designed for, which were open source DBMSs such as PostgreSQL and MySQL. These DBMSs can add clustered and hash indexes after the table was created. Luckily AS3AP accepts standard indexes as substitutions for hash and clustered indexes.

In many DBMSs C APIs database cursors are opened and closed explicitly. This was the case with previous versions of the Oracle Call Interface (OCI), but the new version has cursors that are implicitly opened and closed. This provided a minor problem because there were many calls to functions that did nothing and ended up being a small performance issue because Oracle was opening and closing cursors on every call as opposed to controlled opening and closing of cursors by the application. We also had to fix the cursor fetch function in the Oracle port because the implementation was faulty and returning incorrect data to the calling functions.

The last issue we dealt with was the SQL Loader utility provided by Oracle which is used for moving data from external flat files into the Oracle database. When the application would attempt to populate the database, it was taking upwards of 20 – 25 seconds. This was a huge contrast to the other DBMSs which were taking 3 – 7 seconds to populate the database. With a bit of research we found that using the direct path loading feature of SQL Loader and using TRUNCATE instead of REPLACE in our INTO TABLE clause drastically improved performance by as much as 100% bringing down the populate time to 5 – 7 seconds.

3.2. OSDB Redesign

Due to many of the pitfalls we experienced attempting to complete the Oracle port in C, we thought it would be a good idea to suggest a redesign of the application. The primary goal of the redesign is to make it as easy as possible for developers to add new DBMSs to the test suite and use all existing tests. The secondary goal is to allow the developer to have the ability to easily add new test suites and test cases to the application. We took many of the ideas we learned implementing the Oracle prototype port, and worked off those when developing this preliminary design idea.

3.2.1. Database Class Design

We first created a base class called Database that defines all the methods needed to connect to a DBMS and run any SQL statement. Any class that inherits from this base class will be able to reuse those particular methods.

The syntax for the creation of tables can vary between different DBMSs. In particular, you may wish to express dates, times, floats, or long differently depending on the DBMS vendor. To offer this to the developer we have an abstract base class called DataType which developers can extend and create new data types. Each instance of the Database class has a hash table as a member variable that stores the ways the developer wishes represent each individual data type.

The actual database table is an abstract base class that holds the DataTypes of the table we wish to create. When we want to create a table, we pass in the table object to the Database class' createTable(DatabaseTable) function which determines how to create the table based on the desired syntax for each individual database.

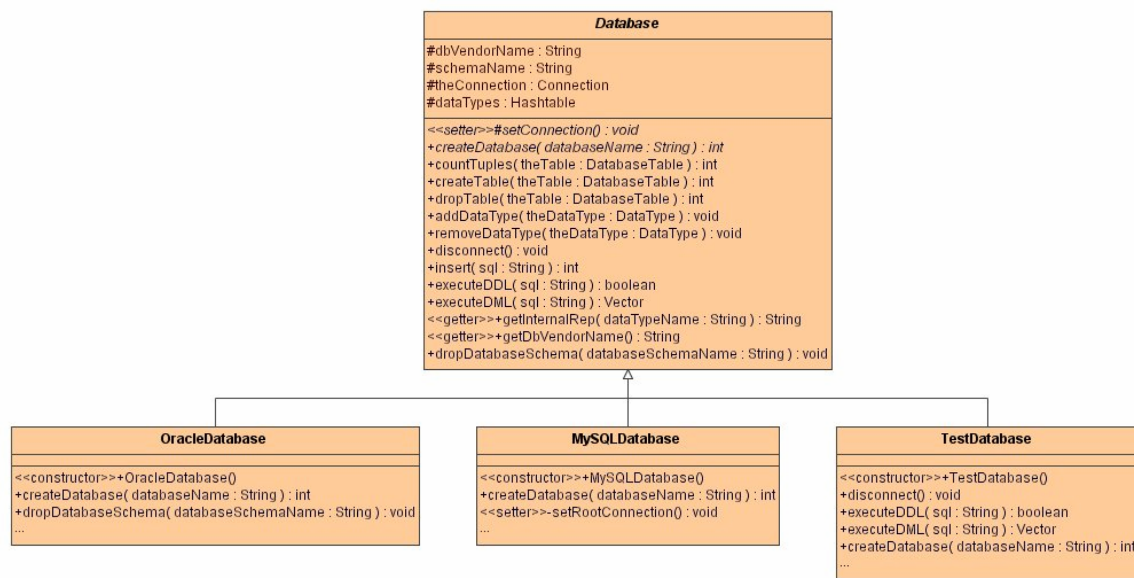

Figure 1: Database Base Class with User Implemented DBMSs

3.2.2. Individual Test Cases

To implement the individual test cases, we decided to use the Command pattern [2]. The Command pattern encapsulates a request as an object allowing us to in a sense, queue each test case for delayed execution. The TestCase class is derived from an interface called Test which contains a single run() function which we will implement very shortly. The TestCase class has two member variables. Those are a simple string that stores the name of the test, such as “Single User Tests” or “Multi User Tests,” and a Timer object that we created that holds a start time and end time for each individual test method in the test case.

3.2.3. Implementing the run() Method

To create standard running behavior for each TestCase, we wanted to create a template for the running of each test method in the TestCase. While running each test, we thought the developer would want to set up and environment such as database data, run the test on that environment, and the tear down that environment much like the JUnit Testing Framework does [5]. To accomplish this we choose to use the Template method which defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. The skeleton calls setup() first, run() second, and teardown() third.

Figure 2: TestCase Class Implementing Test Interface with User Created TestCases

3.2.4. Test Result Reporting

Now that we have our TestCase ready to go, we need a way to report the results of the Tests back to the calling function. We accomplished this by using a common design pattern called a collecting parameter. The Collecting Parameter pattern does exactly as it name says. It gets passed around to the tests and collects all successes, errors, and exceptions. When all the tests in a TestCase are finished running, we can use this collecting parameter class to print us a summary of the TestCase or we can set the collecting parameter class to print us a summary once each test is done.

The collecting parameter class is called TestResult and is a fairly simple class. TestResult has two vectors as member variables that hold on to successful tests and failed tests. We have a class for both failed tests and successful tests. The failed test class, named TestFailure, has three fields; a field that holds the Test class that failed, a field that holds the test name that failed, and the Throwable object that was generated during the test execution. The successful test class, named TestSuccess, has four fields; a field that holds the Test class that succeeded, a field that holds the test name that succeeded, the return value of the test, and the total time it took to run the test. Along with the Vectors of

successful and failed tests, the TestResult class also has a field named totalTests that keeps track of the number of tests the TestResult object has collected.

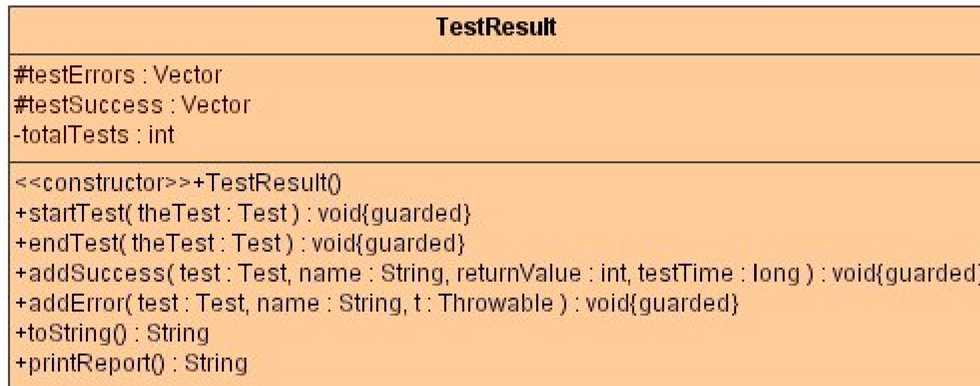

Figure 3: TestResult Class

3.2.5. Error Handling

In order for the TestResult class to properly collect information about if a test fails or succeeds, we need to have some way of handling errors. Error handling is implemented by extending Java's Exception class and creating our own custom Exception called OSDBException. The reason we needed to create our own exception class is because we distinguish between fatal exceptions and non-fatal exceptions. If a fatal exception is thrown, the program cannot continue so we exit.

An example of a fatal exception would be if we could not establish a connection to the database. If that would happen, obviously we would not be able to continue. An example of a non-fatal exception would be a SQL syntax error on one of the tests. Even though the test will fail, we can still continue on and execute the rest of the tests. The fatal and non-fatal exception is distinguished with a Boolean parameter, true being a fatal error and false being a non-fatal error.

To make life easier for the developer, OSDBException is declared abstract to allow for sub-classing. There are four sub-classes of exceptions that could be thrown during program execution. Those are DatabaseException, DataLoaderException, TestCaseException, and TestSuiteException. Each exception can be chained together to allow for an easy back trace.

3.2.6. Defining Test Methods

We now have a way to run a TestCase and collect the information about each test. The next things we need is an easy way for the TestCase's test methods to be run. This

problem is solved using Java's Reflection API. We look through our TestCases for methods that begin with test run those methods, just like JUnit does.

3.2.7. Combining the Tests With Test Suites

In order to run all database tests together in one run, we need to have an object can hold on to the individual TestCase classes and run them in sequence. To accomplish this we can use the Composite design pattern. The Composite design pattern allows us to either treat Tests and single objects (TestCases), or as a group of objects (TestSuites).

The Test interface is used for this purpose. The TestCase classes represent the leaves in the Composite and the TestSuite class represents the composite in the Composite pattern. The run() method then delegates to it's children.

The TestSuite class also implements the Template Pattern by defining setup() and teardown() methods much like the TestCase methods do. This allows the developer to provide functionality of doing specific setup steps before the test is ran, and specific teardown steps after the test has ran.

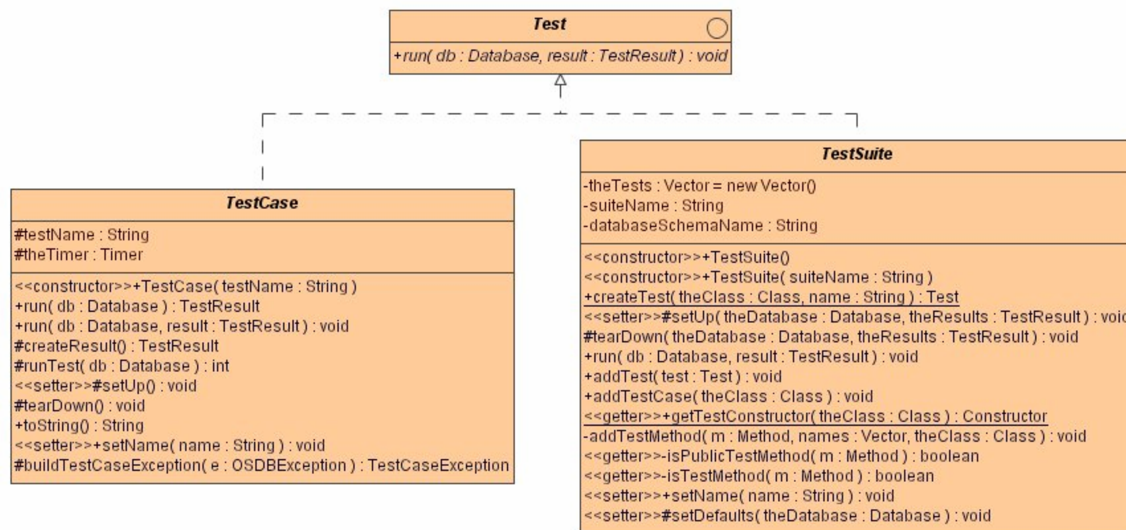

Figure 4: Composite of Test, TestSuite, and TestCase

3.2.8. Putting It All Together

We choose to implement our design in Java. This is primarily because the Java Database Connectivity (JDBC) standard takes care of the implementation details and gives us a nice standard API to use for database connectivity. JDBC provides us the flexibility to declare our methods in our base class for basic behavior such as obtaining a connection to the database, executing DDL and DML statements, and disconnecting from the database.

To pull together our databases, test cases, and test suites we would use the framework in this manner:

```
public static void main(String[] args) throws DatabaseException {

        // parse the command line commands
        parseCommands(args);

        // set the properties for the database
        try {
                loadDatabaseProperties();
        } catch (IOException e) {
                System.err.println("Error loading defaults. " + e.getMessage());
                System.exit(1);
        }

        // create the test suites, etc ....
        TestSuite theSuite = new DefaultTestSuite();
        TestResult testResults = new TestResult();
        Database theDatabase = new OracleDatabase();
        theSuite.addTestCase(SingleUserTest.class);

        // run the test suite
        theSuite.run(theDatabase,testResults);

        System.out.println(testResults.printReport());
}
```

Figure 5: Main Method for Running Tests

4. Evaluation

Consumers of database management systems would greatly benefit from OSDB having a complete Oracle port. Commercial database benchmarking software can cost upwards of $800 per license [9]. For a medium to large size business, an option like this may be possible. For a smaller company, or for personal use, comparing databases using a commercial benchmark tool is not a viable option. However, another option exists for a benchmark tool available in the open source community which is OSDB. OSDB is an open source application that is free to use. Also, Informix is the only commercially supported DBMS for OSDB, yet Oracle has a market share of 41.3% compared to IBMs 30.6% which includes both DB2 and Informix [10]. Completing the port would increase the value and popularity of OSDB as a free benchmarking suite.

Redesigning the OSDB application is beneficial mainly from the developer's standpoint. The extensibility of the C programming language for an application such as OSDB is far from great. To add on any new features, such as a new test, would require modification of the whole application and the build scripts. By redesigning the application using object oriented techniques and design patterns to solve some of the issues would aid the developer in creating domain-specific benchmarks that may be more beneficial to the user than the AS3AP benchmark that is exclusively used in OSDB. Also, using the JDBC API makes almost all database specific operations exactly the same without the developer

having to take the time to learn some of the more cumbersome C APIs. All the developer has to do is utilize the JDBC driver developed by the DBMS vendor.

5. Conclusions

Database benchmarks are important tools in evaluating computer system performance and price / performance for both commercial corporations and single computer users. Finishing up the single user tests for the Oracle port provided us with some great insight into some of the problems that are inherent with the current implementation of OSDB. It also provides a service to all users who are looking for a free but relevant database benchmark to compare DBMSs and computer systems against one another in a scientific way.

Using an object oriented approach to the redesign of the OSDB application provided us with the power to use design patterns to simplify the design and extensibility of the framework, which was an issue with the current OSDB implementation. We used many features of object oriented design such as inheritance which allows us to use many powerful design patterns which were invaluable to us in both developing the application and the capability design patterns have in explaining how to use the framework to other developers who are interested in using OSDB.

6. Future Work

First and foremost, the multi-user AS3AP tests for the Oracle port need to be completed so in the next version of OSDB Oracle can be considered officially supported. This would enquire fixing a bug which fails to implicitly close open cursors in the database, therefore producing a maximum exceeded cursors error. The solution to this problem could be either increasing the amount of possible open cursors in the database, or attempting to explicitly close the cursors when needed.

Secondly, since using JDBC can be slower than using a C API for a DBMS, it would be of great advantage to develop or use a fast and reliable C++ wrapper for each DBMS that OSDB has ports for. The framework could then be implemented using our current design in a language like C++ to provide greater performance gains while running the benchmark, but still keeping to our two primary redesign goals. We also see the benefit of adding a graphical user interface to the benchmark which would allow you to dynamically create databases, tables, tests, test suites, and data types which would allow for an even easier development environment.

To test the new design ideas, it would be helpful to start the development of a prototype benchmarking tool for Firebird DBMS, which was previously commercial, but now is open source. Firebird will also be the first partially object orientated DBMS to have an open source benchmark tool. This will give us more insight into our re-factored software design, and help us design for future object orientated DBMSs.

7. References

[1] J. W. Cooper, *Java Design Patterns: A Tutorial*, Addison-Wesley, 2000.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] J. Gray, *The Benchmark Handbook*, Morgan Kaufmann Publishers, 1991, pp. 167 – 206.

[4] JUnit, Testing Resources for Extreme Programming. http://www.junit.org/

[5] S. Mishra, "Improving SQL* Loader's Performance," May 2001; http://oracle.oreilly.com/news/sqlload_0501.html

[6] Open Source Database Benchmark Project. http://osdb.sourceforge.net/

[7] C. Sliwa, "Linux Starts to Take a More Central IT Role," Feb. 2005; http://www.computerworld.com/databasetopics/data/software/story/0,10801,99901,00.html?SKC=data-99901

[8] Standard Performance and Evaluation Consortium (SPEC), http://www.specbench.com

[9] "Top Database Makers Keep Market Share Slots," USA Today, March 2005; http://www.usatoday.com/tech/techinvestor/industry/2005-03-08-dbms-players_x.htm