2010-12-10

Asynchronous Database Drivers

Michael Adam Heath
Brigham Young University - Provo

Asynchronous Database Drivers

Mike Heath

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Daniel Zappala, Chair
Eric G. Mercer
Bill Barrett

Department of Computer Science

Brigham Young University

April 2011

ABSTRACT

Asynchronous Database Drivers

Mike Heath

Department of Computer Science

Master of Science

Existing database drivers use blocking socket I/O to exchange data with relational database management systems (RDBMS). To concurrently send multiple requests to a RDBMS with blocking database drivers, a separate thread must be used for each request. This approach has been used successfully for many years. However, we propose that using non-blocking socket I/O is faster and scales better under load.

In this paper we introduce the Asynchronous Database Connectivity in Java (ADBCJ) framework. ADBCJ provides a common API for asynchronous RDBMS interaction. Various implementations of the ADBCJ API are used to show how utilizing non-blocking socket I/O is faster and scales better than using conventional database drivers and multiple threads for concurrency. Our experiments show a significant performance increase when using non-blocking socket I/O for asynchronous RDBMS access while using a minimal number of OS threads. Non-blocking socket I/O enables pipelining of RDBMS requests which can improve performance significantly, especially over high latency networks. We also show the benefits of asynchronous database drivers on different web server architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

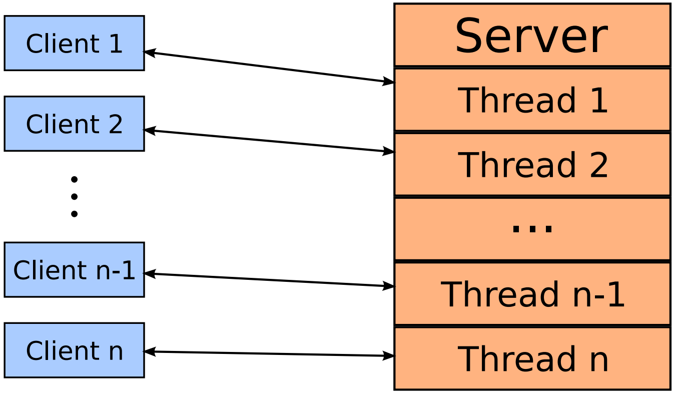
Relational database management systems (RDBMSs) are capable of storing large amounts of data and coordinating access to that data among many concurrent users. Countless business applications, web sites, and other applications rely on RDBMSs to store, organize, and index their data. Whether used for accounting, asset management, or building an Internet auction web site, RDBMSs are the preferred storage mechanism for most business applications built today.

Structure Query Language (SQL) is the lingua franca of RDBMS interaction. SQL can be used for adding, querying, modifying, and removing data from a RDBMS and is a relatively simple data extraction and transformation language. SQL can also be used for defining how RDBMS data should be stored and organized. Applications use a *database driver* to issue SQL statements to the RDBMS. The database driver provides an interface between the application issuing SQL requests and the RDBMS.

Database drivers often use TCP/IP as the underlying network transport to communicate with the RDBMS. To facilitate TCP/IP communication, these drivers typically use blocking socket I/O. Therefore, when an application issues an SQL statement, the thread that calls the database driver blocks until the RDBMS has returned a result. This model is simple for software developers to use and has been employed for many years with great success.

Conventional database drivers work well in a web application that runs on a web server that uses a thread-per-connection architecture. In this type of architecture, each

Figure 1.1: *Thread-per-connection Architecture:* The thread-per-connection architecture is a common architecture used by network servers.



HTTP request is handled by the web server in a separate thread which, in the context of this thesis, is a thread that is known and scheduled by the kernel itself. Because each HTTP request runs in its own thread, when the web applications sends a request to the RDMBS causing the database driver to block, the web server can continue handling other HTTP requests (see Figure 1.1).

Although this combination of a thread-per-connection web server and a blocking database driver works well, researchers and developers are exploring more scalable approaches. The thread-per-connection architecture does not scale well because each thread requires its own stack space, which consumes a significant amount of memory. Additionally, contention for shared resources and the CPU time required to schedule threads can be a significant impediment to a server's ability to scale. As an alternative, developers are building servers using an event-driven architecture, in which multiple network connections are handled by a single thread. This allows the server to handle thousands of concurrent network connections [1].

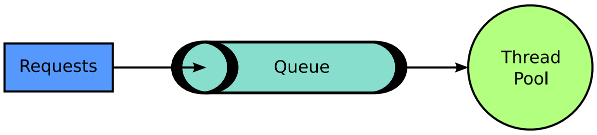
Unfortunately, little work has been done on designing a database driver for this new web server architecture. One of the problems with conventional database drivers is that they do not work well with web servers that use an event-driven architecture because most drivers have a single thread of execution which blocks on every request. As a result, having a single thread manage multiple requests means the thread blocks on every request to the driver.

A second problem with conventional database drivers is that they do not allow for a web application to query multiple RDBMSs in parallel without utilizing additional threads. An example of an application that may need this capability is a web portal. Web portals aggregate information from multiple disparate sources. Such information could include news, stock prices, the weather, and anything else that may be of interest to a particular user. Each RDBMS request must be made sequentially because the database driver blocks. If each RDBMS request takes an average of 200ms, and the portal must make 12 requests, then the web portal will take a minimum of 2.4 seconds to load. This type of delay would make the web portal feel very slow and unresponsive to the user. If each RDBMS could be queried asynchronously, the web portal could respond as quickly as the slowest RDBMS request. This is not a problem limited to web portals. Any time multiple independent RDBMS requests are issued, executing them asynchronously would generally be faster than the conventional sequential approach.

In this thesis, we assert that database drivers using non-blocking socket I/O will increase the performance of web applications that issue many independent requests across multiple RDBMSs and will increase the scalability of database dependent event-driven web applications. To test this, we introduce the Asynchronous Database Connectivity in Java (ADBCJ) project. ADBCJ is a framework for asynchronous RDMBS interaction. Using ADBCJ, we compare two techniques for asynchronously accessing a RDBMS. The traditional approach to asynchronous RDBMS interaction is to push requests onto a queue and process the requests in that queue using a thread pool as shown in Figure 1.2. The alternative is to use non-blocking socket I/O.

Our results show that using non-blocking I/O is faster than the thread-pool approach for many common RDBMS interactions. Non-blocking I/O also enables the use of pipelining to improve the performance of issuing multiple requests to an RDBMS, especially over high

Figure 1.2: *Thread pooling:* Requests are enqueued to be executed by an arbitrary thread within a pool of threads.



latency connections. Additionally, we demonstrate the use of ADBCJ in a simple web application that shows its benefits when accessing multiple RDMBSs, regardless of whether non-blocking I/O or a thread pool is being used to parallelize database calls. Finally, we demonstrate how well a web-server using an event-driven architecture is able to scale when used in conjunction with an asynchronous database driver using non-blocking I/O.

In summary, this thesis makes the following contributions:

- A database driver API for asynchronous database interaction in ADBCJ
- An ADBCJ database driver that uses JDBC to communicate with the RDBMS and a thread-pool for asynchrony
- ADBCJ drivers for MySQL and Postgresql that use non-blocking socket I/O for asynchrony
- A demonstration of the performance benefits of pipelining RDBMS requests
- A performance comparison of using a thread-pool versus non-blocking socket I/O
- A scalability analysis of our different database drivers when used on an event-driven web server

Additionally, the database drivers we produced have been used at eBay for load testing and are being used by LinkedIn.

Chapter 2

Related Work

A great deal of research has been dedicated to database drivers, event-driven networking, and web server performance analysis. This section provides an overview of some of this research and highlights some of the important factors we need to consider.

2.1 Existing Database Drivers

Many software platforms have well defined APIs for RDBMS interaction. The software that implements these APIs for a particular RDBMS are called database drivers. These APIs provide a common interface that is independent of the actual RDBMS software (i.e. MySQL, Oracle, Postgresql, etc.) that is being used. The Windows operating system has ODBC [2], the Java platform provides JDBC [3], Perl has DBI [4], Python has its own database driver API [5] and the list goes on. These database driver APIs all require that each request block until the RDBMS has processed the SQL request and returned a response.

A popular technique to achieve asynchronous RDBMS access is to use thread pooling (see Figure 1.2). With thread pooling, individual requests are placed in a queue, allowing the requesting thread to return immediately without blocking. A pool of worker threads services the requests in the queue. The thread pool can process as many concurrent RDBMS requests as there are threads in the pool. The Twisted framework is an example of a framework that supports this technique for asynchronous RDBMS access [6]. The web application has to be built to take advantage of thread pooling.

Repeated RDBMS interaction can be improved by pooling RDBMS connections. Connection pooling preserves connections between requests, allowing them to be reused thereby eliminating having to needlessly duplicate the work required to connect and authenticate to the RDBMS. Connection pooling not only improves performance but has also been shown to improve web application scalability in some cases. If the web application acquires and releases an RDBMS connection from the pool on an as needed basis, the number of requests being processed at one time can exceed the number of RDBMS connections available in the pool. If the web application acquires an RDBMS connection from the pool when it begins processing a web request and returns it when the request is finished, no other web requests may use that connection. Exclusively holding an unused RDBMS connection prevents other resources from using this limited resource. Conversely, if the web application acquires an RDBMS connection only when it needs to query the database and immediately returns the connections to the pool when the query has been processed, the web application can continue doing non-database work while a different request uses the now available connection [7].

2.2 Web Server Architectures

Internet servers must be able to service multiple clients concurrently. There are various architectures that can be used to achieve this. Some of the more popular server architectures that are commonly used when building web servers are outlined below.

2.2.1 Thread-Per-Connection

As stated in the introduction, the most common approach to building network servers in general and web servers in particular is the thread-per-connection architecture. By default, system calls to read and write to a network socket are blocking calls. If a thread is waiting for data from one socket, it cannot do anything else until the read completes. To deal with multiple sockets, we need multiple threads. In the thread-per-connection architecture, there is one thread for each socket being processed. When the threads blocks to perform I/O on the socket, the blocking does not affect the other sockets being processed by different threads.

To minimize the cost of creating and destroying threads, many existing servers maintain a pool of threads for socket I/O. This is the same technique we described earlier in this section for achieving asynchronous RDBMS interaction. Thread pooling makes it possible for the server to reuse threads which is faster than creating a new thread for each new connection. An additional benefit of thread pooling is that the server can accept more connections than there are threads in the thread pool. This is possible because new connections are placed in a queue before being processed by a thread. If all the threads in the pool are in use, the socket is not processed until a thread in the pool becomes available. This will increase the latency required to process the connections but in many cases increased latency is better than rejecting all incoming connections when the server is under heavy load. Servers that use thread pooling also have to be careful not to fill their queue to the point that they run out of memory.

The problem with the thread-per-connection architecture is that threads are a relatively expensive operating system resource compared to a network socket. Each thread requires its own stack, which consumes memory. If a shared resource is required by each thread, contending for exclusive access to that shared resource can have a significant impact on performance. Additionally switching back and forth between many different threads both consumes CPU cycles and reduces the effectiveness of the CPU's second-level cache because the CPU has to continually address so many different stacks.

The Apache HTTP Server 2.0 is an example of a thread-per-connection web server. To support different operating systems as optimally as possible, Apache uses a modular design called Multi-Processing Modules (MPMs) to support different threading models. Various MPMs are bundled with Apache. The *prefork* MPM forks a process to handle each HTTP request. This is similar to using multiple-threads but can be more stable and makes it possible to use libraries that are not thread-safe. The *worker* MPM spawns multiple processes each of which manages its own pool of threads. The worker MPM is faster than the prefork MPM but the prefork MPM is used by default on Unix operating systems for backwards compatibility with Apache HTTP Server 1.3. There are also MPMs that take advantage of platform specific threading or networking features. The MPM used by Apache may be configured during compilation or at run-time [8].

2.2.2 Event-Driven

The problem of scaling to thousands of concurrent network connections has been outlined by Kegal [1]. The prevailing solution is to use an event-driven architecture. In an event-driven architecture, a single thread can process hundreds or even thousands of concurrent network connections. Because event-driven servers do not not have to deal with the added overhead incurred by using a large number of threads, event-driven architectures can scale better and handle a significantly greater number of concurrent network connections.

In a server that uses an event-driven architecture, each thread indicates to the operating system which network sockets it is interested in. When the state of a socket the thread is interested in changes, the thread is notified of that change in the form of a socket event. The socket event indicates which socket had a state change, what the change was, and any data that may be needed to process the event. Examples of socket state change include the acceptance of a new socket connection by a listening server socket, receipt of new data on a socket, or a socket closure.

When data is written to a socket, it is done asynchronously. This means that the thread won't block when it writes data to the socket. The data to be written is stored in an operating system buffer until the data has been successfully sent to the remote host. The operating system buffers are typically limited in size, which prevents the threads from sending large amounts of data in a single request. If the thread needs to send more data to a socket than the operating system is able to buffer, it must send a portion of the data and wait until the operating system indicates that more data may be sent to the socket. These write availability notifications come in the form of socket events the same way the thread receives other socket events.

Event-driven architectures are much more complex than thread-per-connection architectures for a number of reasons. In a thread-per-connection architecture, each network request is handled in a straightforward sequential fashion. In an event-driven architecture, each thread has to correlate each socket event to a particular request and maintain each request's state in its own state machine. Additional complexity comes from reading fragmented network data. Let's say, for example, that a server is expecting 1000 bytes of data. In a thread-per-connection architecture the server can block until all 1000 bytes have been received. In an event driven architecture, the server may receive multiple data read events containing less than the desired 1000 bytes. It is the server's responsibility to accumulate these events until the desired amount of data has been received.

Fortunately there are various frameworks that help to reduce the complexity of event-driven application development. Some of these frameworks include: Apache MINA [9], Grizzly [10], and Twisted [6]. These frameworks automatically handle certain types of socket events, alleviating the need to manually do so by application developers. They also provide tools for maintaining the state of different network connections, assist in assembling fragmented network data, as well as facilitate sending large amounts of data.

On POSIX based operating systems, the system calls *poll()* and *select()* can be utilized for event-driven networking. *poll()* and *select()* both work by accepting a list of sockets as arguments and return to the caller which socket events need to be processed. Banga et al. demonstrate how this approach is not very scalable when working with thousands of socket descriptors [11]. Both *poll()* and *select()* have to iterate over the list of socket descriptors presented to them as arguments to determine which socket has pending events. The Banga et al. paper introduces explicit event notification, in which an application registers with the operating system which socket descriptors it is interested in only once. The application can then request events for the registered sockets. This eliminates the need to repeatedly pass a list of socket descriptors to the operating system. It also makes it easier for the operating system to be fair in determining the order in which an application receives socket events.

The explicit event notification system described by Banga et al. has been implemented for the Linux platform using *epoll* [12], for the Solaris platform using */dev/poll* [13], and for the various BSD platforms using *kqueue* [14]. Each explicit event notification implementation has been shown to be faster and more scalable than their *poll()* or *select()* counterparts.

There are various types of event-driven architectures. Some of these architectures are described briefly here. Pariag et al. provide a more detailed explanation and of each architecture [15]. Their paper compares the performance and scalability of event-driven architectures with thread-per-connection architectures. They report that although the different event-driven architectures have different performance and scalability characteristics, they consistently outperform and scale better than even the cutting edge thread-per-connection architecture.

2.3 Measuring Web Server Performance and Scalability

One of the widely used benchmarking suites for web server performance is SPECweb [16]. There are various versions of SPECweb; the latest of which is SPECweb2005. SPECweb2005 provides both a Java Server Pages (JSP) and a PHP based application that are used for measuring a particular web server's performance. This application emulates the functionality of an e-commerce application. The SPECweb2005 load generator produces workloads that are representative of real user access patterns. SPECweb2005 also simulates the behavior of popular web browsers by downloading images using two parallel HTTP connections and simulating browser caching.

However, there are two significant problems with SPECweb2005 for our purposes. The first problem is that the provided server application is not built for an event-driven web server. The second problem is that the provided application is not capable of effectively using asynchronous event-driven database drivers. An additional problem is that the

SPECweb2005 load-generator client does not generate overload conditions because it is a *closed-loop* system [17]. In a closed loop system, the client does not begin a new request until the previous request has finished. This restriction prevents the client from issuing more requests than the server can reliably handle. The solution is to use a partially open-loop system in which the number of requests sent to the server is a set amount and any requests that do not complete within a predetermined amount of time are closed and marked as failures. *httperf* [18] is a tool that creates a partially open-loop system and is the tool used by Pariag et al. [15].

Chapter 3

Solution

3.1 ADBCJ

To compare different approaches for asynchronous RDBMS interaction, we built *Asynchronous Database Connectivity in Java* (ADBCJ). ADBCJ is similar to existing database drivers in that it provides an abstract API to define how database interaction is to be done. The key difference between the ADBCJ API and traditional APIs is that calls to ADBCJ that have the potential to block return immediately and execute asynchronously. This means that the calling thread does not have to wait for the database operation to complete and can instead perform other operations.

ADBCJ follows a *service provider interface* (SPI) design similar to many other database driver APIs. The SPI design provides a system wherein multiple *service providers* implement a common *API.* Consumers of the API need not worry about which implementation is in use as long as they follow the contract defined by the API. In the context of RDBMS interaction, service providers are commonly referred to as database drivers. The SPI design allows us to build a single benchmark and run the benchmark using different implementations of the API and compare the results without having to make modifications to the benchmark [19].

3.2 ADBCJ API

The ADBCJ API defines an interface for achieving RDBMS-independent connectivity similar to what is currently provided by existing APIs such as JDBC, ODBC, etc. The primary tasks that are facilitated by this API are (1) establishing a connection to an RDBMS, (2) sending

SQL statements to the RDBMS without blocking, and (3) notifying the application through an event dispatch mechanism when the result of an SQL statement is ready for consumption.

Currently the ADBCJ API only provides support for basic database functionality. We currently support querying and updating the database using resource local transactions. ADBCJ currently only supports simple data types such as integers, floating points, and strings. Providing a complete API with support for things like late binding, binary objects, distributed transactions, etc. would go beyond the scope of this thesis.

The ADBCJ API is flexible enough to be used in both a blocking manner and an asynchronous event-driven manner. When an SQL statement is sent to the server, the API immediately returns a *future* object [20], which represents a pending operation. A future object promises to make the result of the pending operation available when the operation completes. In the case of an error, the future object is used to propagate the error.

When using ADBCJ, any operation that has the potential to block will instead execute asynchronously to the calling process and return a future object. The future object can be used to cancel the pending request or block until the pending operation has completed. Additionally, ADBCJ future objects are also *observable* as described by the *observer design pattern* [21]. This means that one or more *observers* (or event handlers) can be registered with the future object and the observers will be called when the database operation has completed. The listener gets called when the database operation completes. If a listener is added to the future after the operation has completed, the listener gets invoked immediately by the thread that is adding the listener.

The observable future approach makes ADBCJ flexible in how it is used. ADBCJ can be used in a fork-join fashion. In fork-join, the calling thread initiates the execution of several asynchronous tasks and then waits for all the tasks to complete. Psuedo code for this can be done using ADBCJ is shown in Figure 3.1. Alternatively, ADBCJ can be used in an event-driven approach as shown in the psuedo code in Figure 3.2.

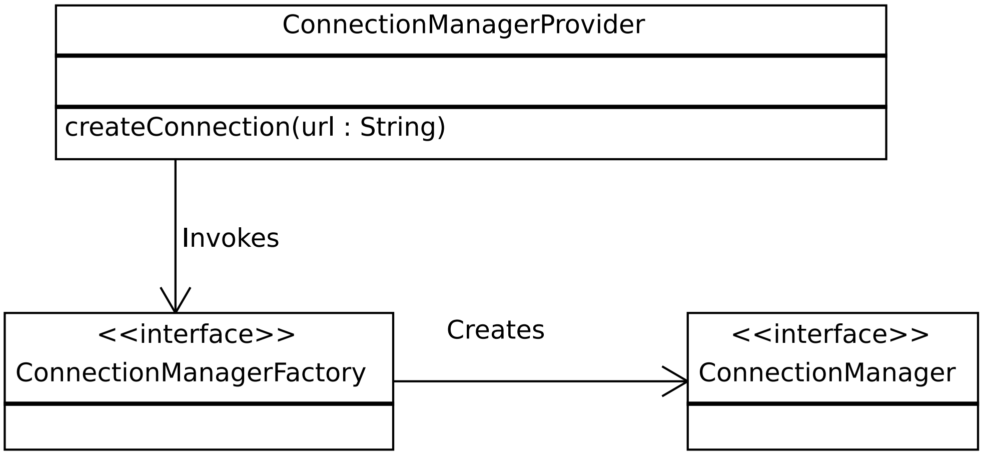
Figure 3.1: *Fork/Join:* This Java code snippet shows how ADBCJ can be used for doing *fork/join* style concurrency.

```java
Future<ResultSet> future1 = connection.executeQuery(...);
Future<ResultSet> future2 = connection.executeQuery(...);
Future<Result> future3 = connection.executeUpdate(...);
ResultSet rs1 = future1.get();
ResultSet rs2 = future2.get();
Result result = future3.get();
```

Figure 3.2: *Event-driven:* This Java code snippet shows how ADBCJ can be used for *event-driven* style concurrency.

```java
connection.executeQuery(...).addListener(
        new DbListener<ResultSet>() {
                public void onComplete(Future<ResultSet> future) {
                        ...
                }
        }
);
connection.executeUpdate(...).addListener(
        new DbListener<Result>() {
                public void onComplete(Future<Result> future) {
                        ...
                }
        }
);
```

Figure 3.3: *ADBCJ SPI Relationship:* Each ADBCJ driver must provide a *ConnectionManagerFactory* instance that is used by the *ConnectionManagerProvider* to provide a *ConnectionManager* instance to the application.



3.2.1 Connecting to an RDBMS

To use ADBCJ, an application uses the class *ConnectionManagerProvider* to obtain an instance of ConnectionManager. A *ConnectionManager* instance is what is used to connect to an RDBMS. To create a *ConnectionManager* instance, the application must provide a *connection URL*. The connection URL is a RFC 1738 compliant URL [22]. The URL scheme is always "adbcj". A sub-scheme follows the scheme indicating which driver the application wants to use. The host and port portions of the URL are naturally used to determine which host to connect to and which port the RDBMS is listening on for connections. The path portion of the URL indicates which database schema the application wants to use. A username and password may also be specified in the URL to authenticate the application to the RDBMS. An example ADBCJ URL might look like "adbcj:mysql://localhost:1368/test" where "mysql" indicates that a MySQL driver is to be used to connect to a MySQL instance running in the local machine and will use the "test" database. JDBC uses a similar scheme for indicating which database driver should be used.

The relationship between *ConnectionManagerProvider*, *ConnectionManagerFactory*, and *ConnectionManager* is show in Figure 3.3.

Figure 3.4: *ADBCJ Application:* A sample ADBCJ application executing an SQL query against a MySQL database.

```
ConnectionManager connManager = ConnectionManagerProvider
                .createConnectionManager("adbcj:mysql://example:password@localhost/example");
DbFuture<Connection> connectFuture = connManager.connect();
Connection connection = connectFuture.get();
DbSessionFuture<ResultSet> resultSetFuture = connection.executeQuery("SELECT␣field␣FROM␣example");
resultSetFuture.addListener(new DbListener<ResultSet>() {
        public void onCompletion(DbFuture<ResultSet> future) throws Exception {
                ResultSet rs = future.get();
                for (Row row : rs) {
                        System.out.println(row.get("field"));
                }
        }
});
```

The *ConnectionManager* is used to connect to the RDMS and manages all the established connections to the RDBMS. This operation could block because connecting to an RDBMS involves network I/O. Therefore, the *ConnectionManager* returns a future object that is used to obtain the instance of *Connection* that represents the RDBMS connection. The *Connection* object is what is used to send SQL queries to the database.

3.2.2 Result Sets

SQL query results are stored in *ResultSet* objects. A *ResultSet* is a collection of rows and a row is a collection of values associated with a specified database column. The *ResultSet* interface in ADBCJ extends the *java.util.List* interface from the Java collections API. This makes it easy for application developers to access each row in the result set.

By default ADBCJ will return up to 1,000 rows. This is done to prevent the application from running out of memory if a very large result set is returned by the RDBMS.

3.2.3 Sample Usage

A sample application that uses ADBCJ to connect to a MySQL RDBMS and issue a simple SQL query is shown in Figure 3.4.

Lines *1-2* show the creation of a *ConnectionManager* instance. The example ADBCJ URL on line *2* indicates that the program wants to use a MySQL driver. To connect to the RDBMS, the URL specifies that the username ”example” should be used with the password ”password”, the application should connect to the local host, and the MySQL database ”example” should be used.

Lines *3-4* establish a connection to the RDBMS. The call to the *connect()* on line *3* returns a future object. On line *4* the application blocks until the connection is established.

On lines *5* a query is asynchronously sent to the RDBMS and the value of the returned future object is held in the *resultSetFuture* variable. Instead of blocking, as we did when connecting, we add a listener to the *resultSetFuture* on line *6* that will be invoked when the result is received from the RDBMS. The future object that is passed in to the listener on the *onCompletion* method on line *7* is the same future object that is stored in the variable *resultSetFuture*. On line *8*, we extract the *ResultSet* instance from the future object. Lines *9-11* iterate over the results and print the results out to the console. We can use the Java for each loop syntax on the result set because of the fact that a *ResultSet* is a Java list.

3.3 ADBCJ Database Drivers

For this thesis, we built various ADBCJ database drivers. One driver uses a thread pool for asynchrony and delegates RDBMS commands to a conventional JDBC driver. We compare the performance of the thread pool approach against various ADBCJ drivers that use non-blocking I/O for asynchrony.

3.3.1 Thread Pool Database Driver

The thread pool based database driver uses the same approach that we described earlier in 1.2. Any RDBMS request that has the potential to block is queued up and a thread in the pool picks up the request, executes it using a conventional blocking JDBC driver, and puts the result in the request’s future object. If the future object has any event listeners registered with it, the event listeners are invoked by the pooled thread that made the call to the RDBMS using JDBC.

The Java platform has built in support for thread-pools using the *ExecutorService* API. This API provides out-of-the-box support for creating thread-pools and submitting tasks to the thread-pool. The *ConnectionManager* instance that is created when using this driver holds the *ExecutorService* instance and uses this *ExecutorService* instance for executing RDBMS connection and query operations.

The ADBCJ URL for using this driver starts with ”adbcj:”, as is required by all ADBCJ URLs. The remaining part of the URL is a conventional JDBC URL. For example, the URL ”adbcj:jdbc:mysql://example:password@localhost/example”, indicates that the thread pool database driver should be used and in turn the JDBC MySQL driver should be used for the actual RDBMS interaction.

The most challenging aspect of building our thread pool based database drivers is that because we rely on JDBC for the actual communication to the RDBMS we have to deal the limitation of the JDBC *Connection* not being thread safe. A single JDBC *Connection* instance can be accessed by different threads, this just can not happen simultaneously. Therefore, we cannot simply submit RDBMS requests to the connection pool as they arrive. Instead, our database driver maintains its own queue. The request at the head of the queue gets submitted to the thread pool. When the submitted request completes, the driver checks if there is a request waiting in its queue and submits it to the thread pool for execution.

3.3.2 Non-blocking I/O Database Driver

The non-blocking database drivers required us to implement the specific RDBMS over wire protocols from the ground up as it is not possible to take an existing database driver and simply tell it to start using non-blocking I/O. The RDBMSs that we decided to support are MySQL and Postgresql. We chose these RDBMSs because they are open source and their network protocols are well documented.

To implement the drivers, we looked at two different non-blocking I/O frameworks: Apache MINA[1] [9] and JBoss Netty [23]. Both are high performance non-blocking I/O frameworks both designed to make implementing network protocols easier.

MINA and Netty both implement the *reactor pattern*. The reactor pattern is a design pattern that describes how multiple concurrent inputs are demultiplexed and dispatched to request handlers. This pattern is used by many non-blocking I/O frameworks because it provides a clear separation of protocol logic and I/O handling.

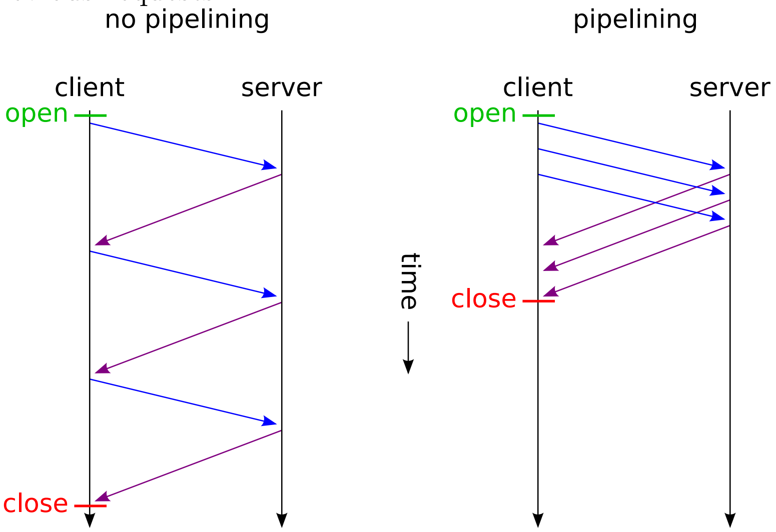
After doing some performance analysis [24], we chose to use Netty. One of the key benefits of Netty that makes it well suited to our purposes is that Netty can send network data from the calling thread without that calling thread blocking. When sending data, Netty checks if the socket's write buffer is full. If the buffer is not full, the data can be sent immediately. If the buffer is full, the data gets queued to be sent later. This gives Netty clients many of the same performance characteristics that blocking I/O clients have. Additionally, Netty uses a number of sophisticated lock-free data structures that are better suited for high scale environments.

One of the significant benefits of non-blocking I/O is that it allows us to pipeline requests to the RDBMS. This optimization has produced dramatic benefits. Pipelining provides the ability to send more than one request without waiting for the response from previous requests. This concept is illustrated in Figure 3.5. We examine the benefits of pipelining more deeply in Section 4.1.2

One of the challenges of building an asynchronous system like ADBCJ is managing queue sizes. Even with non-blocking I/O and pipelining, we still have to queue up requests until the response for the request is received. If we queue up too many requests, we will eventually run out of memory. Clearly the size of the queue has to be capped at some point. ADBCJ has two configurable approaches for dealing with a queue that is at its limit. The first approach is to immediatly fail and throw an exception. The second approach is to block

[1] The author was a member of the Apache MINA Project Management Committee.

Figure 3.5: *Pipelining:* Using pipelining, multiple requests may be sent without waiting for responses from previous requests.



until the queue size has decreased. Some thread pool implementations have a third option to deal with large queues, and that is to execute the task to be queued from the calling thread. This approach violates the *first in first out* properties of a queue and is not acceptable for ADBCJ because RDBMS requests must often be executed in order for a given connection.

Chapter 4

Results

By doing a few micro-benchmarks, we show the relative performance of using blocking JDBC drivers, non-blocking drivers using JDBC and a thread-pool, and using a driver using non-blocking I/O. Our non-blocking drivers are able to significantly outperform blocking JDBC drivers, especially over high latency connections, due to the ability to pipeline multiple requests. We also show the performance characteristic of using our various database drivers within the context of a conventional Java web server accessing multiple data sources. Finally, we examine the impact on scalability of our different database drivers when using an event-driven web server.

All of our benchmarks are run using the Oracle Java SE Runtime Environment build 1.6.0_19-b04 using the Java HotSpot Server VM. Benchmarking Java applications can be tricky because the HotSpot VM can execute Java byte code in an interpreted mode or convert the byte code to instructions native to the underlying processor. Converting the byte code to native code is not a static process either as the code can go through various levels of optimization depending on how frequently it gets executed. A naive benchmark running on HotSpot would simply run the code being measured multiple times and average the results. The problem with this approach is that the first few executions of the code could be interpreted, subsequent requests could be compiled to native code, and if the code is executed frequently enough, the native code could be further optimized.

To get around the optimization problem, it is recommended that Java based benchmarks go through a *warm-up* first and then go through a *measurement phase*. The rec-

Table 4.1: *Performance Comparison of Various Database Drivers*

MySQL		
Driver	Mean	Std. Dev.
JDBC	23ms	2.7ms
Thread-pool	34.9ms	10.5ms
Netty	4.84ms	1.6ms

Postgresql		
Driver	Mean	Std. Dev.
JDBC	25.9ms	11.5ms
Thread-pool	79.2ms	12.7ms
Netty	5.65ms	1.0ms

ommendation from Oracle is to execute the code being measured at least 50,000 times to ensure that it gets fully optimized by the HotSpot VM. In all of our benchmarks, we execute the code to be measured 100,000 times to ensure that it is fully optimized and then take measurements [25].

Unless otherwise stated, all of our benchmarks were run on machines with a single Intel Xeon dual core processor running at 2.33 GHz with 2GB of RAM running on Linux 2.6.21.7-2 SMP from Red Hat. Each host is on the same gigabit/second switched Ethernet network.

4.1 Micro Benchmarks

4.1.1 How does the performance of our different database drivers compare?

First and foremost, we compare the difference in performance between conventional blocking I/O JDBC drivers, our ADBCJ drivers that execute JDBC calls asynchronously using a thread-pool, and our non-blocking I/O drivers using Netty. This benchmark executes the same SQL query 50 times. For the conventional JDBC test, we ran each query sequentially. For the ADBCJ tests, we scheduled each query to be executed and then used the return future objects to block until all the requests had completed using a fork/join approach for concurrency as described in Chapter 3.2.

This benchmark was run on one host that connected to a second host running either MySQL or Postgresql over a 1Gb/s Ethernet network. We used a sample size of 1,000 and show the average query response time in Table 4.1.

Analysis

We anticipated that the performance of conventional JDBC would be faster than using a thread-pool because of the overhead of queuing/dequeuing requests and the subsequent inter-thread communication required for notifying the calling thread that the request has completed. However, we did not expect this overhead to be as significant as it is. Additionally, we did not expect such a big difference between MySQL and Postgresql. After doing some profiling and deeper investigation, we have no explanation for this disparity. We ran similar benchmarks executing queries with varying result set sizes as well as different SQL command such as *INSERT*, *UPDATE*, and *DELETE* SQL operations that all resulted in relatively similar results.

When using Netty, the performance difference is significant. This is primarily due to the ability to pipeline requests when using non-blocking I/O.

4.1.2 What are the performance benefits of pipelining?

Pipelining provides significant performance benefits, and we wanted to examine at what point does pipelining impact performance in a positive way. To do this we built an experiment that sent multiple requests over the same connection. We started with a single request and moved up to 25 requests. We ran the experiment using the loopback interface, over a gigabit/second Ethernet network, as well as over the Internet[1]. To compare the difference, we used a simple JDBC driver as a baseline, and then use our non-blocking driver with pipelining disabled, and again with pipelining enabled. We collected 1,000 samples for each configuration.

[1] The hosts we used in our Internet experiment were connected using a Comcast cable Internet connection in West Jordan, Utah through the BYU network connecting to a database located at the Riverton Office Building of the LDS Church.

Conclusion

We show the average latency of each experiment in Figures 4.1, 4.2, and 4.3. The latency over the loopback device is extremely small and, as would be expected, there is no significant difference between pipelining vs. non-pipelining on MySQL. On Postresql, however, we see a small benefit is gained from pipelining. The latency on a gigabit/sec Ethernet network is still relatively low, but even after just two requests, there is a significant improvement in performance. Over the Internet, where latency is relatively high, we see an improvement by a factor of five.

4.1.3 How do our ADBCJ drivers scale?

We look at how well ADBCJ scales both horizontally and vertically. Horizontal scaling demonstrates how well ADBCJ performs across multiple RDBMS hosts while vertical scaling demonstrates how well ADBCJ performs with multiple connections to a single host.

One of the goals of ADBCJ is to be able to efficiently interact with a large number RDBMSs. To measure how well ADBCJ is able to do this, we created an experiment that connects to an arbitrary number of RDBMSs and executes a single SQL query that returns a small result set. We chose to use a single query because we did not want any advantages from pipelining to bias the results. We measured the time it takes to asynchronously send a query to each RDBMS and receive a response using fork/join style concurrency. We ran the experiment multiple times. We first connected to a single RDBMS and then sequentially increased the number of RDBMSs to eight. We would have tried connecting to more RDBMS hosts if we had the available hardware to reliably run the experiment. We executed our test query 50,000 times for each configuration and show the averages in Figure 4.4. Ideally, as the number of RDBMSs increase, the graph should remain relatively flat. When using non-blocking socket I/O, the slope of the graph is fairly shallow. The graphs when using a thread pool are not as shallow but still scale fairly well.

Figure 4.1: *Pipelining results over loopback interface:* When the latency is very low, there is no clear performance increase from pipelining when using MySQL and a slight performance gain for Postgresql.

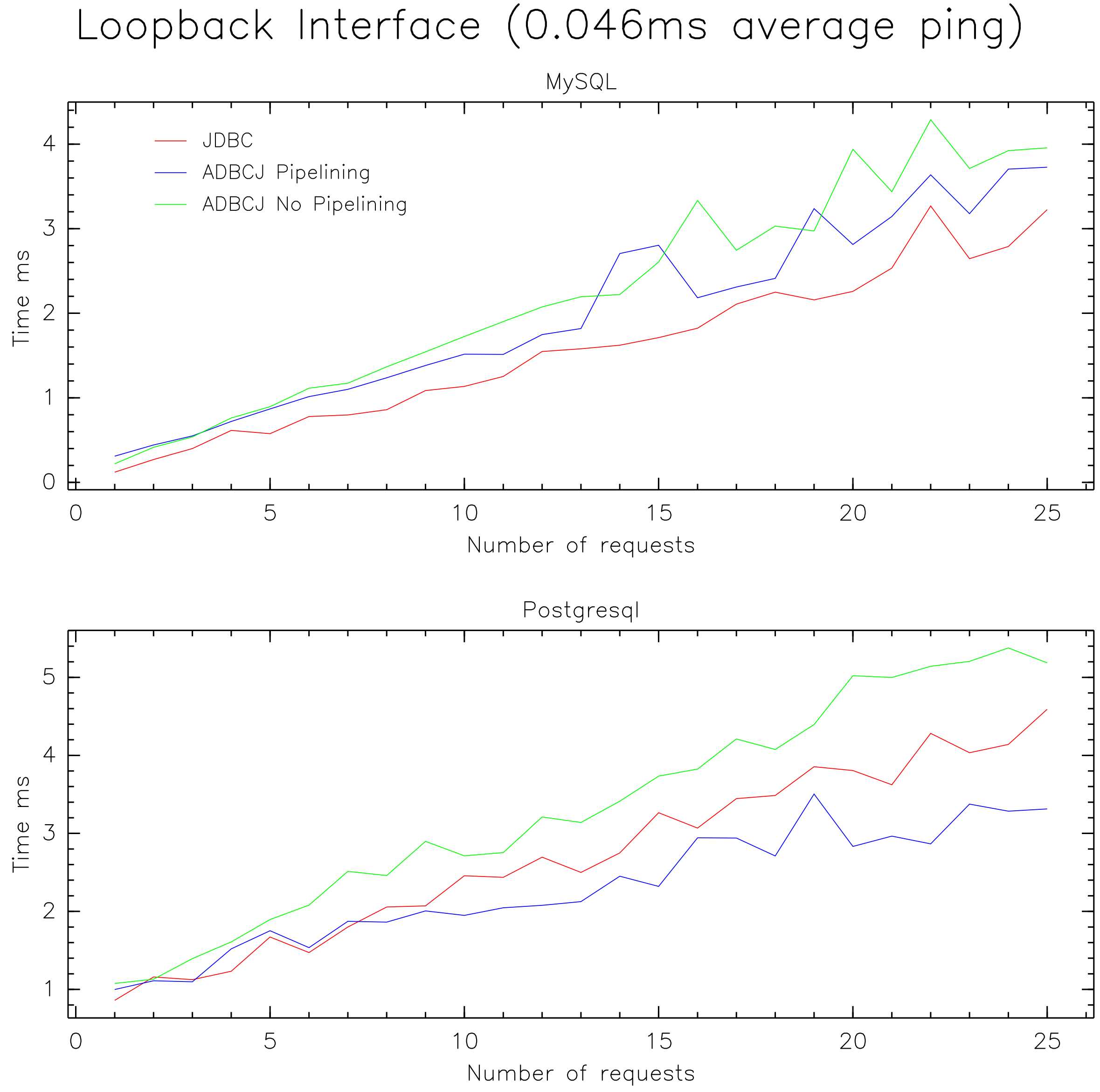


Figure 4.2: *Pipelining results over 1 gigabit/sec network:* Even with moderately low latency on a gigabit/sec Ethernet network, using pipelining has a three times performance improvement.

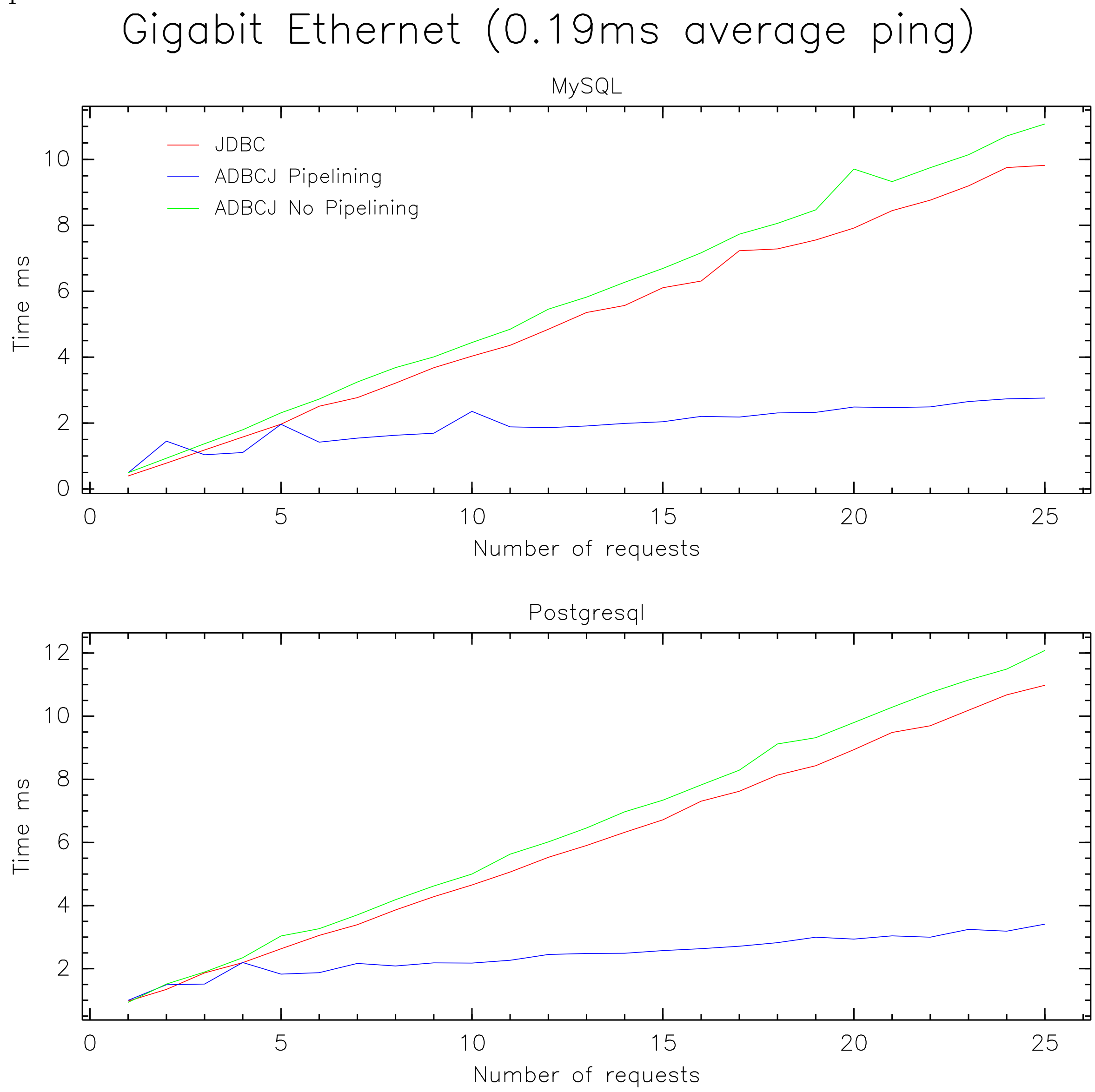


Figure 4.3: *Pipelining results over the Internet:* When the latency is relatively high, using pipelining produces a five times perfmance increase.

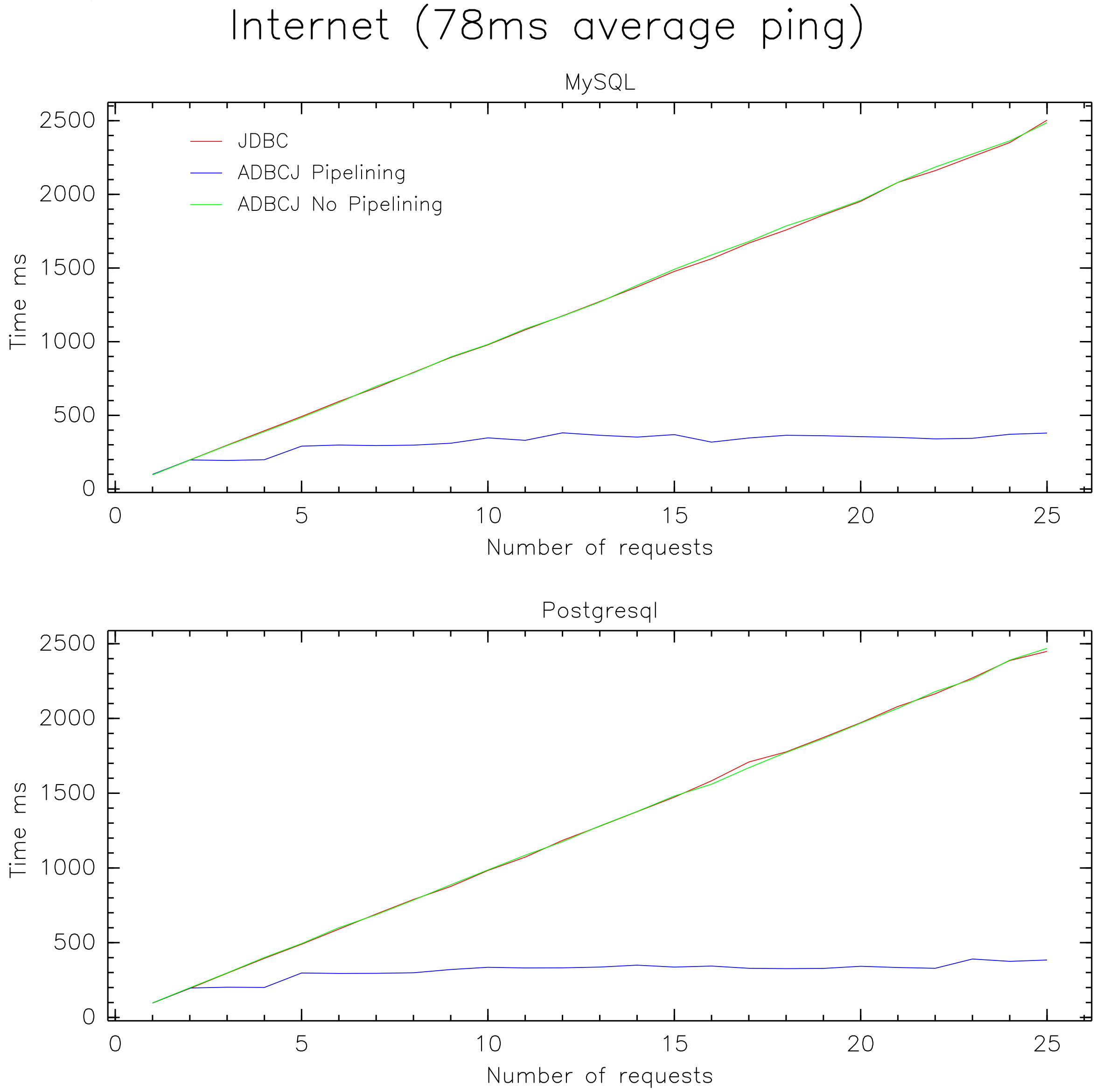


Figure 4.4: *Horizontal scalability:* Horizontal scaling results.

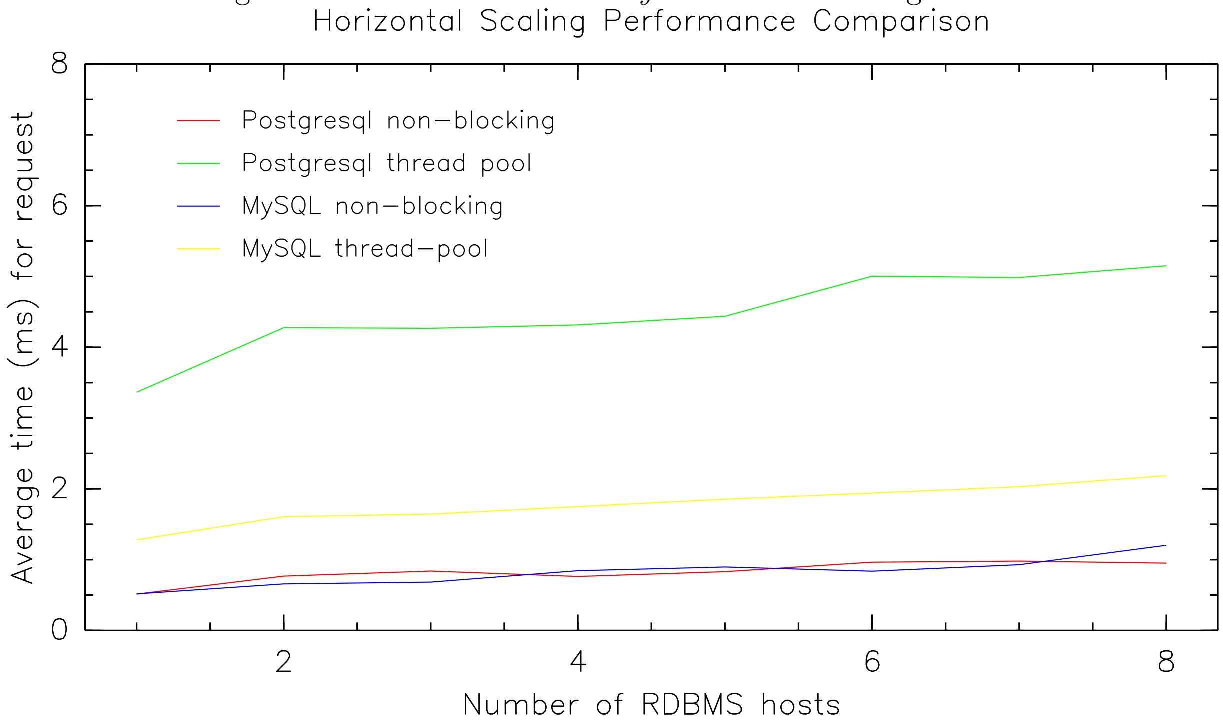
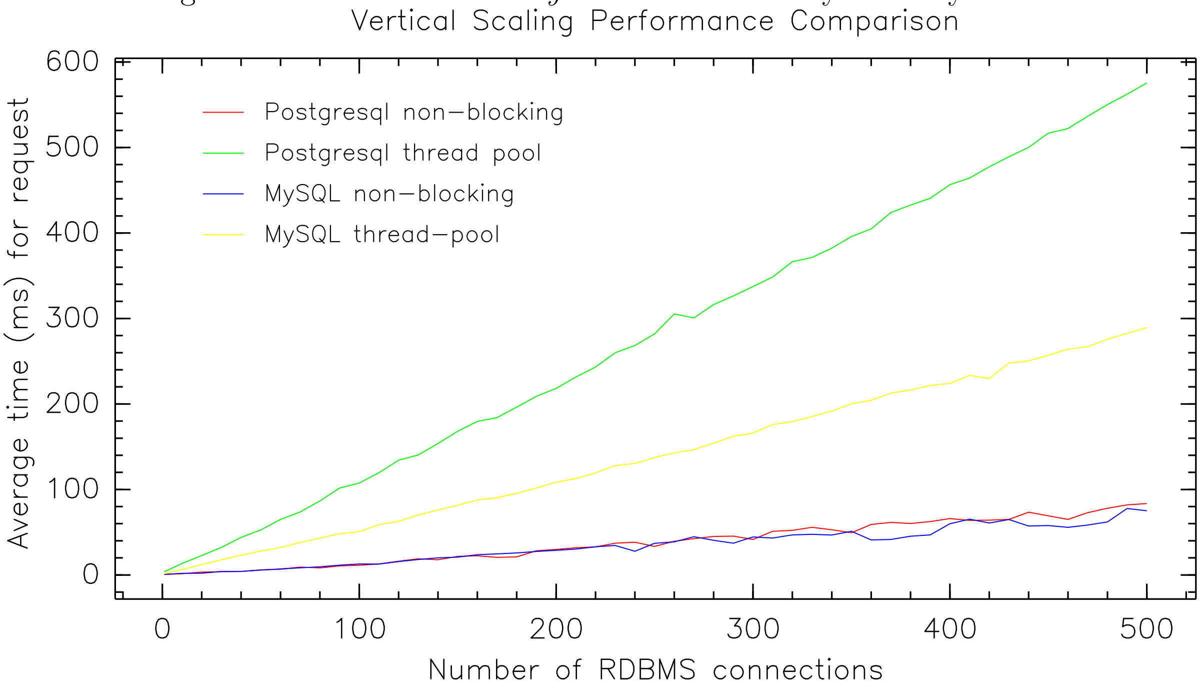


Figure 4.5: *Vertical scalability:* Vertical scalability is nearly linear.



Vertical scalability is a challenging issue due to the fact that the RDBMSs we are using cannot scale to a high number of concurrent connections. By default, both MySQL and Postgresql have a limit of 100 concurrent client connections. Increasing the number of connections that MySQL can handle is simply a matter of changing the MySQL configuration property *max_connections*. The maximum value of this property is, according to MySQL documentation, limited by the available hardware [26].

Scaling Postgresql is more difficult. MySQL uses a thread-per-connection architecture whereas Postgresql forks new processes and uses shared memory for coordinating the various processes. For details on how we were able to scale Postgresql, see Appendix A.1.

To measure the vertical scalability of ADBCJ, we built an experiment that creates a specified number of connections to the RDBMS. We then measure the time it takes to send the same SQL query we used in the horizontal scalability experiment across each connection and get a response. The results are shown in Figure 4.5. The difference between the non-blocking socket I/O results and the thread pooling results are more dramatic than the horizontal scaling results. As the number of connections increases, the difference in performance is much less with non-blocking socket I/O than with thread-pools.

Analysis

As we increase the number of RDBMSs, performance degrades more slowly giving non-blocking socket I/O an advantage for horizontal scalability. Our non-blocking socket I/O drivers also have a clear performance advantage when vertically scaling up to many connections to a single RDBMS instance.

4.2 ADBCJ on a Conventional Web Server

One of the use cases that we identified when initially developing ADBCJ was a portal like application that aggregates data from multiple RDBMSs. We predicted that asynchronously fetching the data would be faster than serially fetching the data. To test this hypothesis, we

Figure 4.6: *Database Driver Performance Comparison Rate:* The average number of requests per second for each load (higher is better).

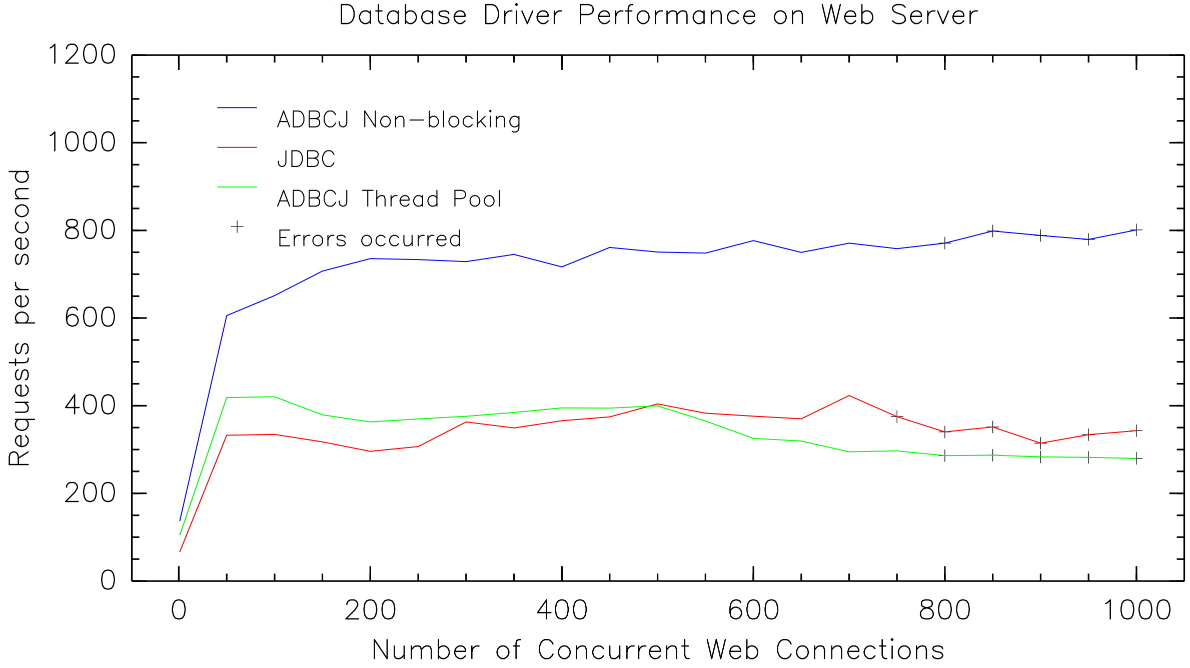
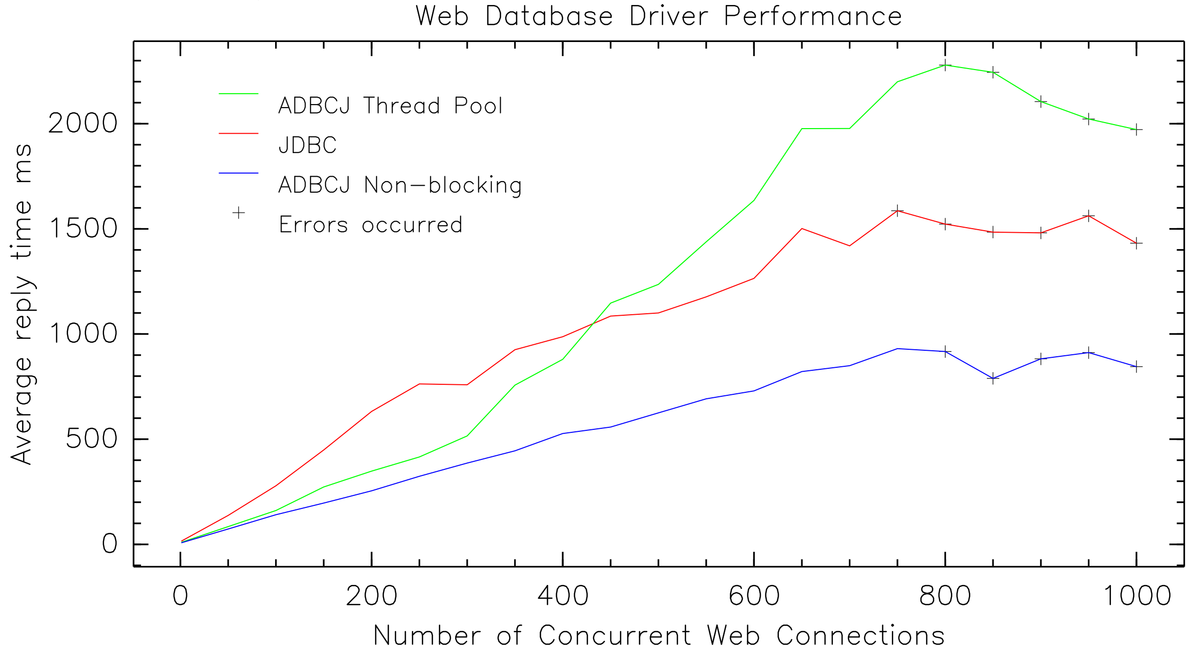


Figure 4.7: *Database Driver Performance Comparison Reply Time):* The average reply time in ms for each load (lower is better).



designed a portal like web application that aggregates data from multiple sources. We built two versions of this application, one that uses conventional JDBC and another that uses ADBCJ. We compare the performance of serially fetching the data versus various techniques for asynchronously fetching the data.

We chose two datasets for this test. One dataset is a list of 40,000 names and phone numbers. The second dataset contains 5 million rows of web access logs. We put each of these datasets on separate MySQL and Postgresql instances, resulting in a total of four data sources from which to aggregate data.

Our application queries 5 contacts at random from each contact database. The application queries the access logs over a one hour period, selected at random, and returns the number of times a given URI was accessed during that period.

We built our application using the Java Servlet API. The Java Servlet API is the standard API for building web applications on the Java platform [27]. We deployed the various versions of our application to the Apache Tomcat 6.0.26 servlet container [28]. See Appendix A.3 for details on how we changed the default Tomcat configuration.

To test the performance of our application, we used httperf to run various tests ranging from 1 connection to 1,000 concurrent HTTP connections, in increments of 50. Each connection connected 100 requests. We measured the number of requests/second the server was able to handle as well as the number of threads and memory usage of the web server. The parameters we used for executing httperf are found in Appendix A.2.

Our application renders a single HTML page containing the query results. To generate this page, we used the FreeMarker template engine library [29]. Both versions of our application use the same FreeMarker template.

The JDBC version of our web application simply executes each query serially in a manner conventional to many database driven web applications. Because Tomcat does not provide database connection pooling support, we used a connection pool provided by the Spring Framework [30].

The ADBCJ version of our web application uses the fork-join approach to concurrency similar to what is shown in Figure 3.1. We tested this application in two configurations: one using our thread-pool based drivers and the second using our non-blocking I/O drivers. To pool the ADBCJ RDBMS connections, we used a simple connection pooling mechanism that we built into ADBCJ.

One of the challenges of using thread-pools is determining how many threads to use in the pool. The more threads we have, the more concurrent requests we are able to process. With our configuration, we were able to create about 2,200 threads in the JVM before getting *OutOfMemoryExceptions*. Tomcat needs 1,000 of these threads for handling HTTP requests. It also uses a handful of threads for monitoring and management activities. So, for the tests that use the thread-pool based ADBCJ driver, we used a thread-pool with a maximum of 1,100 threads.

The initial results for our thread-based asynchronous database drives were abysmal, averaging around 36 requests/second. Our earlier performance tests involved either a single connection and multiple queries or multiple connections and a single query. It turns out that when multiple queries are issued with multiple connections, performance degrades significantly. In this experiment, we issue multiple queries to the RDBMSs managing the contacts databases. The first query is promptly submitted to the thread pool. Once the initial request has completed, the subsequent request is submitted to the thread pool. At this point, the queue to the thread pool has grown and the latency required to execute the subsequent requests increases as well. This queuing behavior is by design to ensure fairness in the case of a large number of connections trying to use a small thread pool.

To solve the latency problem, we added a mode of operation to our thread pool-based asynchronous database driver that lets it operate in an "unfair" mode. When a RDBMS request completes execution in the thread pool, it checks to see if there are subsequent requests for the connection. If so, the requests are executed immediately. We use the "unfair" mode of operation in this experiment.

With the systems we used for testing, a user can have up to 1,024 file descriptors open. Each TCP socket uses a file descriptor. With Tomcat using 1,000 file descriptors for handling HTTP requests and each request using 4 file descriptors to communicate with each RDBMS host, we had to increase this limit. See Appendix A.5 for details on how we changed the file descriptor limit.

We ran our experiment using six individual hosts. Four hosts for running the various RDBMSs, one host running Tomcat, and a third host for running httperf. In Figure 4.6 we show the results of the experiment, graphing the average number of requests/second against the number of concurrent web connections. In Figure 4.7 we show the average reply time for each request.

The thread pool-based database drivers can query each RDBMS concurrently, outperforming the JDBC database drivers when the number of web connections is low. However, as the number of concurrent web requests increases past 400, the thread pool gets saturated, thereby increasing the latency of queued requests and the performance starts to degrade.

Our non-blocking I/O database drivers, however, improve performance dramatically. We believe the primary reason for the dramatic performance difference is that these drivers are able to send data to the socket from the calling thread. This eliminates the latency incurred when queuing up requests to be executed in a second thread. We are also able to use pipelining and benefit from its performance gains when issuing multiple queries to the contacts databases.

As the load increases, the driver starts encountering errors. All of the errors we logged stem from connection reset errors communicating with one of the Postgresql RDBMs. After getting a connection reset error, we saw numerous connection timeout errors trying to reconnect to the Postgresql instance.

Figure 4.8: *Event-driven Web Server Performance Requests/Second:* The average number of requests per second for each load (higher is better).

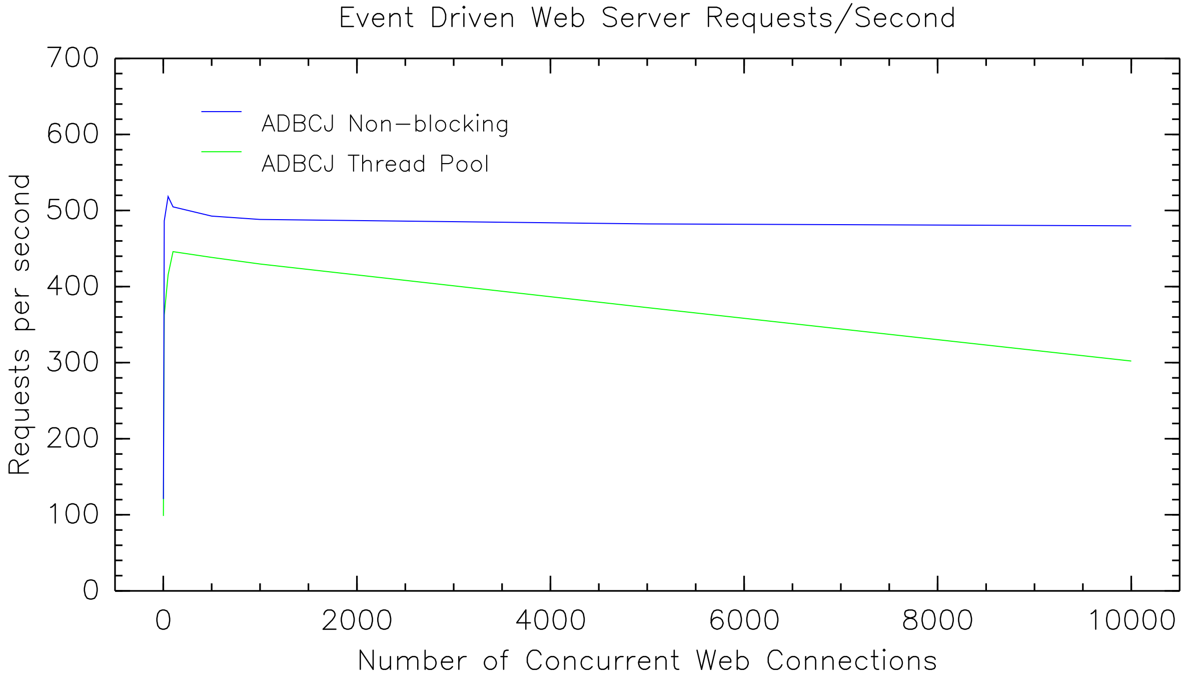
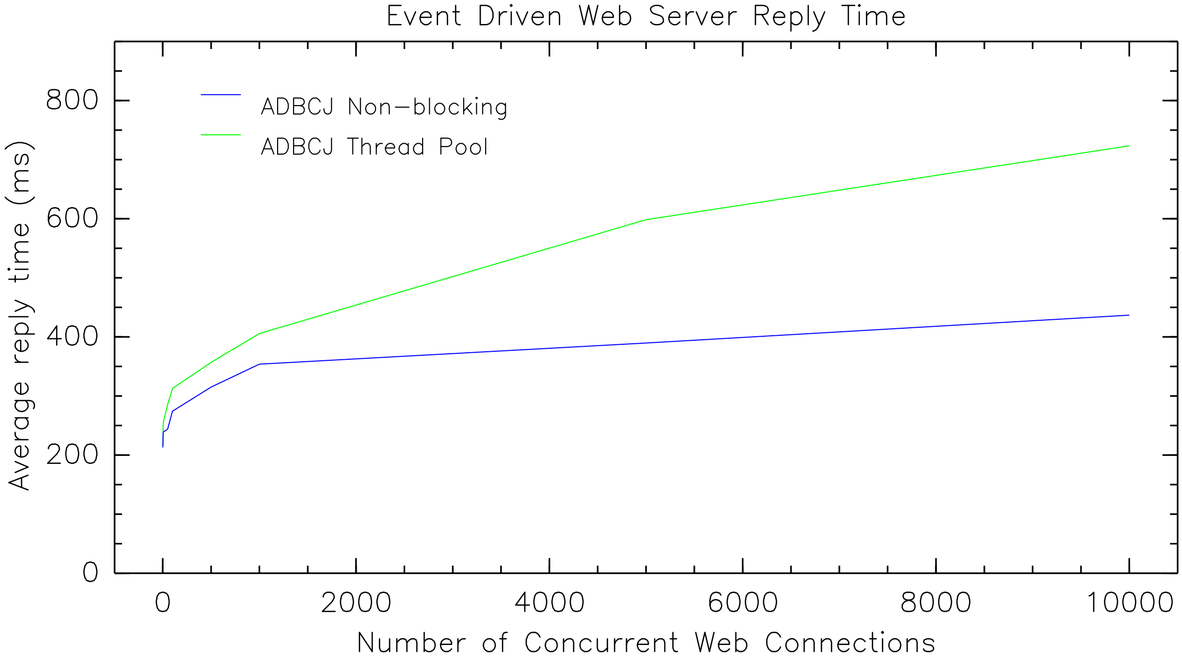


Figure 4.9: *Event-driven Web Server Performance Reply Time:* The average reply time in ms for each load (lower is better).



4.3 ADBCJ on an Event-Driven Web Server

On the Java platform, there are various frameworks for building event-driven web servers, but they are different and do not adhere to common standard like the Servlet API. Among these various frameworks, we chose to use Netty to build our event-driven web server. This was an obvious choice as we are already using Netty for our database drivers.

The event-driven web application we built simulates the functions that a real e-commerce application might have including searching for items, viewing the details of an item, and purchasing an item.

Our goal is to be able to scale this application to 10,000 concurrent connections. This was a difficult challenge given the fact that relational databases themselves typically do not scale well. With the hardware that we used for our other tests, we were unable to get Postgres to scale reliably past 700 or connections. MySQL was able to scale past a few thousand concurrent connections but this was still inadequate. For this test, we were able to secure a server with a quad-core Intel Xeon processor running at 2.66GHz with 8GB of RAM running the 64-bit Linux 2.6.21.7-2 SMP kernel from Red Hat. On this machine we were able to scale MySQL to 10,000 concurrent connections but could not scale Postgresql reliably past a few thousand concurrent connections. Because of this, we chose to only use MySQL for this experiment.

Similar to our experiment on a conventional web server, we chose to use FreeMarker for rendering HTML. This application has five simple web pages for doing the following:

- A landing page that allows the customer to search for items. The search terms are chosen by random from a pre-determined list of search terms.
- Search results.
- Item details.
- A purchase item page.
- A purchase successful page.

We used three hosts to conduct this experiment: the large database server mentioned earlier, a host for running the event-driven web server, and a host for running httperf. On all three hosts we had to change the file descriptor limit as we did for the conventional web server experiment. Additionally, httperf by default is configured to handle a maximum of 1,024 connections. To remedy this, see Appendex A.4 for details on how we modified httperf to scale beyond 1,024 connections. The parameters we used for running httperf are found in Appendix A.2.

To run httperf, we used the same parameters as in the previous experiment (see Appendix A.2) except for the –*uri* parameter. We chose to always have httperf use the same URI to make the httperf configuration simpler. The logic that gets invoked from this URI determines which page should be rendered and sent back to the client.

For the thread-pool based database drivers we use a pool of 2,200 threads. Because our event-driven web server uses a small number of threads, we can use a much larger thread-pool than in the previous experiment.

We did not run any plain JDBC experiments on our event-driven web server. Because the JDBC calls block, we know that this would perform poorly given the fact that our web server has so few threads.

We ran httperf various times going from a single connection up to 10,000 concurrent connections. In Figure 4.8 we show the results of the experiment, graphing the average number of requests/second against the number of concurrent web connections. In Figure 4.9 we show the average reply time for each request. The thread-pool based driver still scales fairly well however it is clear that the non-blocking I/O drivers are able to scale better and have less performance degradation as the load increases.

4.4 Threats to Validity

Our micro-benchmarks are by no means exhaustive. Our drivers only support simple types such as *varchar* and *int*. We did not add support to our drivers for more complicated types such as those for handling large amounts of text or binary data. The results of interacting with these types of data could be very different from the simple data we examined.

All of our pipelining experiments executed simple SQL queries. More complicated queries may take longer to execute and this longer execution may make the performance improvements from pipelining negligible.

When running our web based experiments, we only used the Apache Tomcat web servers. However, there are many Java servlet containers and other implementations of the servlet specification may yield different performance results than the ones we gathered. Given the simplicity of our test application, we feel that the results would be similar.

Our experiments run on an event-driven web server are the most suspect since there are many different event-driven web server frameworks. We believe that our results would be similar across different web frameworks. However, the fact that we used a Netty based database driver and a Netty based web server may have impacted our results.

All of our experiments were run on Linux. Given that different operating systems implement asynchronous I/O differently, our results may vary across different platforms. Future research should look at comparing and contrasting our results across multiple operating systems.

Chapter 5

Conclusion

This thesis presents a performance and scalability oriented comparison of thread-pool and non-blocking I/O asynchronous database drivers using the ADBCJ asynchronous database driver framework. We show that using non-blocking socket I/O for asynchronous database drivers has many advantages over using a thread-pool with conventional database drivers. Queries can be sent to the RDBMS immediately instead of waiting to be executed in a second thread, improving performance significantly. Multiple requests can be pipelined which also improves performance in most cases. Non-blocking socket I/O based database drivers also scale better.

We show how asynchronous database drivers can improve the performance of web applications that access multiple datasources. The performance of asynchronous database drivers that use non-blocking I/O instead of thread-pools are also faster in this situation. We also show that using non-blocking I/O in conjunction with an event-driven web server scales better than using a thread-pool for asynchrony.

However, asynchronous database drivers have a significant drawback in that they are much more complicated to use than conventional blocking database drivers and dealing with concurrency can be very difficult.

A future version of Java will have a new API for doing asynchronous I/O [31]. This new API, to a large degree, mitigates a lot of the advantages of frameworks like Netty by providing a concise API for asynchronous I/O. It would be interesting to see how a database driver built on this API would perform compared to our existing driver built on Netty.

Database drivers are only one area where the benefits of non-blocking socket I/O has been demonstrated. Future research should be done to determine how non-blocking socket I/O could be used with emerging non-relational databases such as Cassandra [32], CouchDB [33], MongoDB [34], Voldemort [35], and other so called *nosql* databases.

References

[1] D. Kegel, “The C10K Problem,” 2006. [Online]. Available: http://www.kegel.com/c10k.html

[2] Microsoft, “ODBC–Open Database Connectivity Overview,” 2007. [Online]. Available: http://support.microsoft.com/kb/110093

[3] “JDBC 4.0 API Specification Final Release,” 2006. [Online]. Available: http://java.sun.com/products/jdbc/download.html

[4] “Perl DBI,” 2007. [Online]. Available: http://dbi.perl.org/

[5] P. D. SIG, “Python Database API Specification v2.0,” 2006. [Online]. Available: http://www.python.org/dev/peps/pep-0249/

[6] T. M. Labs, “Twisted Matrix Labs: Building the engine of your Internet,” 2007. [Online]. Available: http://twistedmatrix.com/

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel, “Performance and Scalability of EJB Applications,” *SIGPLAN Notices*, vol. 37, no. 11, pp. 246–261, 2002.

[8] Apache Software Foundation, “Apache HTTP Server Version 2.2 - Multi-Processing Modules (MPMs),” 2007. [Online]. Available: http://httpd.apache.org/docs/2.2/mpm.html

[9] ——, “Multipurpose Infrastructure for Network Applications,” 2010. [Online]. Available: http://mina.apache.org/

[10] Sun Microsystems, “Project Grizzly,” 2007. [Online]. Available: https://grizzly.dev.java.net/

[11] G. Banga, J. C. Mogul, and P. Druschel, “A Scalable and Explicit Event Delivery Mechanism for UNIX,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1999.

[12] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, “Comparing and Evaluating epoll, select, and poll Event Mechanisms,” in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, 2004, pp. 218–228.

[13] S. Acharya, “Using the devpoll (/dev/poll) Interface,” Sun Microsystems, Tech. Rep., 2002. [Online]. Available: http://access1.sun.com/techarticles/devpoll.html

[14] J. Lemon, “Kqueue - A Generic and Scalable Event Notification Facility,” in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2001, pp. 141–153.

[15] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, “Comparing the Performance of Web Server Architectures,” *ACM SIGOPS Conference on Computer Systems*, vol. 41, pp. 231–243, March 2007.

[16] S. P. E. Corporation, “SPEC Benchmarks - Web Servers,” 2005. [Online]. Available: http://www.spec.org/benchmarks.html#web

[17] G. Banga and P. Druschel, “Measuring the Capacity of a Web Server,” in *Proceedings of the USENIX Symposium on Internet Technologies and Systems.* Berkeley, CA, USA: USENIX Association, 1997.

[18] D. Mosberger and T. Jin, “httperf a Tool for Measuring Web Server Performance,” *SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[19] J. Bloch, *Effective Java.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[20] H. Baker Jr and C. Hewitt, “The Incremental Garbage Collection of Processes,” *ACM SIGART Bulletin*, pp. 55–59, 1977.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[22] M. M. T. Berners-Lee, L. Masinter, “Uniform Resource Locators (URL),” 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1738.txt

[23] JBoss, Inc., “Netty Project,” 2010. [Online]. Available: http://www.jboss.org/netty

[24] T. Lee, “Performance Comparison between NIO Frameworks,” 2008. [Online]. Available: http://www.jboss.org/netty/performance/20081006-tlee.html

[25] M. Paleczny, C. Vick, and C. Click, “The Java Hotspot(TM) Server Compiler,” in *JVM’01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium.* Berkeley, CA, USA: USENIX Association, 2001.

[26] Oracle, “MySQL 5.1 Reference Manual:Too Many Connections,” 2010. [Online]. Available: http://dev.mysql.com/doc/refman/5.1/en/too-many-connections.html

[27] Java Community Process, “JSR 154: Java Servlet 2.4 Specification,” 2004. [Online]. Available: http://jcp.org/en/jsr/detail?id=154

[28] Apache Software Foundation, “Apache Tomcat,” 2010. [Online]. Available: http://tomcat.apache.org/

[29] Visigoth Software Society, “FreeMarker: Java Template Engine Library,” 2009. [Online]. Available: http://freemarker.sourceforge.net/

[30] Spring Source, “Spring Framework,” 2010. [Online]. Available: http://www.springframework.org/

[31] Java Community Process, “JSR 203: More New I/O APIs for the JavaTM Platform (”NIO.2”),” 2007. [Online]. Available: http://jcp.org/en/jsr/detail?id=203

[32] Apache Software Foundation, “Apache Cassandra,” 2010. [Online]. Available: http://cassandra.apache.org/

[33] ——, “Apache CouchDB,” 2010. [Online]. Available: http://couchdb.apache.org/

[34] 10gen, Inc., “mongoDB,” 2010. [Online]. Available: http://www.mongodb.org/

[35] O. P. V. c. LinkedIn, Inc., “Project Voldemort,” 2010. [Online]. Available: http://project-voldemort.com/

Appendix A

Experiments

A.1 Scaling Postgresql

To increase the number of client connections that Postgresql can handle we must update its *max_connections* property. We must also update the amount of shared memory that Postgresql will use. The version of Postgresql we are using requires about 128KB of shared memory per connection. However, our Linux installation has a default maxed shared memory setting of about 33.5MB, limiting our maximum number of connections to about 262. Fortunately we can update the maximum about of shared memory the Linux kernel will allow by writing the desired maximum to the file */proc/sys/kernel/shmmax*.

A.2 httperf Parameters

When running our experiments against a conventional web server, we used the following httperf configuration paramters:

–hog This tells httperf to to not limit itself to using ephemeral ports. Staying within the limited range of ephemeral ports can often be a bottleneck.

–server Used to indicate the IP address of the web server.

–port The port the web server is listening on. In our case, using Tomcat, this defaults to 8080.

–uri The URI to be requested. The value of this parameter changed depending on whether or not we were using the JDBC version of the application or the ADBCJ version.

–num-calls The number of requests to be issued per connection.

–num-conns The number of connections to use. We started at 1 connection and increased to 1,000 connections in increments of 50.

–rate The rate at which connections are crated. We used the same value here as we did for the *–num-conns* parameters so that all the connections would be used concurrently.

A.3 Tomcat Configuration

By default, Tomcat will scale to 200 concurrent connections. To allow Tomcat to handle 1,000 concurrent connections, we modified the file *conf/server.xml* and changed the *Connector* element for port 8080 by adding *maxThreads="1000"*. We also increased the maximum heap size for the JVM running Tomcat to 1GB.

A.4 Scaling httperf

By default, httper only scales to 1,024 concurrent connections. To remedy this, we modified the header file "/usr/include/bits/typesizes.h" and change the line "#define __FD_SET_SIZE 1024" to "#define __FD_SET_SIZE 65536". We then downloaded the source code to httperf and compiled it. Our modified httperf is now able to scale well beyond what we needed to complete our experiments.

A.5 Changing File Descriptor Limit

We chose to increase the limit to 65,536 just to be on the safe side. To do so we appended the lines _* hard nofile 65536_ and _* soft nofile 65536_ to the file "/etc/security/limits.conf".

To verify that the change actually took place, we executed *ulimit -n*.