ACHIEVING GREAT WEB APPLICATION PERFORMANCE USING ONLY SQL AND PL/SQL

Dr. Paul Dorsey, Dulcian, Inc.

Introduction

Dulcian was tasked with building the budget and finance system for the Finance Ministry of Ethiopia. Not surprisingly, this was a large and complex system. However, in many ways, a description of this system is not significantly different from other systems. There were thousands of users spread across hundreds of locations throughout the country. The transaction load was large and there were hundreds of reports to support.

One less common factor of this system was the requirement to support up to 20 different languages (some of which required Unicode to represent the character set).

The project involved replacing a well built legacy Microsoft SQL Server Java EE system with an Oracle-based system. What helped to make the project even more challenging was the need to avoid having to retrain all users, which meant making no changes to the user interface, even though a complete database redesign was necessary.

The Challenges

One of the biggest challenges was the need to support an infrastructure where download speeds average 5-20K/second and most user locations have no internet connectivity whatsoever. Ethiopia is a large country (twice the size of Texas). Most locations outside of the regional capitals cannot be reached on paved roads.

From a staffing perspective, building systems in the developing world is more challenging than building similar systems in a developed nation. There is rarely ready access to experienced Java EE talent. Even though IT professionals in developing nations are as qualified as in other parts of the world, there is usually a very high turnover rate (particularly in the government). This makes using a complex Java EE-based development environment unsuitable.

In this case, even though the legacy system was fairly well designed, a decision had been made not to implement referential integrity in the database. As a result, over time, direct database fixes resulted in massive amounts of dirty data in the source systems. Numerous instances of orphaned records made data migration particularly challenging.

In addition, cultural and language differences between foreign and local staff and the government created even more logistical difficulties for the project staff.

The Problems of Existing Development Frameworks

There are two significant issues drawbacks to using existing development frameworks:

1. They are too complex.
2. They require too much bandwidth.

The complexity issue is particularly disturbing. Fifteen years ago it was possible to design and develop a complex system using Oracle Forms, Oracle Reports, and Oracle Designer. A single designer could build a system on a home computer, copy it onto a floppy disk, take it to a client site, and install the system in a few minutes. Nowadays, with all of the diverse Java EE architectures, a team of experts is required to design and develop a system. Installation is also a non-trivial task. Such architectures take many months to master, and the rapid continual evolution makes much of the information obsolete in a very short time period. This level of complexity is a difficult issue that frustrates many organizations in the developing world. The higher turnover, smaller Java talent pool, and greater constraints on resources make even usage of Oracle's ADF (arguably the best, simplest, most mature Java EE architecture) difficult, if not impossible.

The bandwidth issue is just as problematic. With the trend towards moving to more and more sophisticated, rich AJAX applications, page size expands to megabytes. In high band width areas where 1MB/second or better speeds are common this

is not an unreasonable requirement. However, in places where bandwidth is limited (which also includes much of rural North America), page sizes over 1MB are unusable.

Oddly enough, web architects tend to live in a fantasy world where bandwidth is nearly limitless. They seem to think that users want objects that spin around on the screen while waiting patiently for the page with all of its beautiful AJAX components to load.

Figure 1 shows how most web architectures communicate. A request starts at the client, moves from the application server to the database, back to the application server and finally returns to the client. Unfortunately, more information than is necessary moves between components and, in many cases, requires multiple round trips between components.

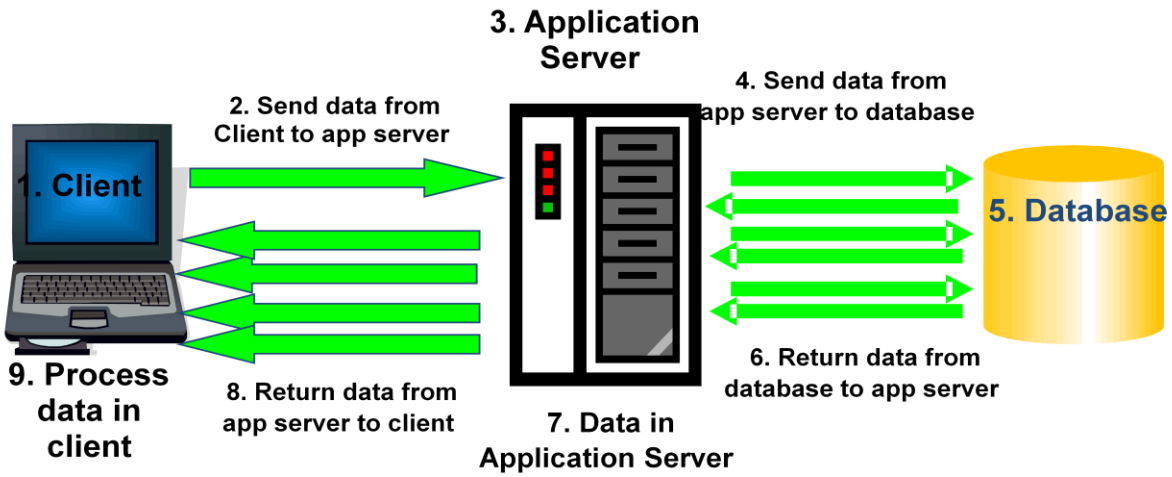

Figure 1: Web Architecture Communication

Based on a review of a number of poorly performing web applications, it can be concluded that most performance problems come from the same sources. There sources are proportionately diagrammed in Figure 2.

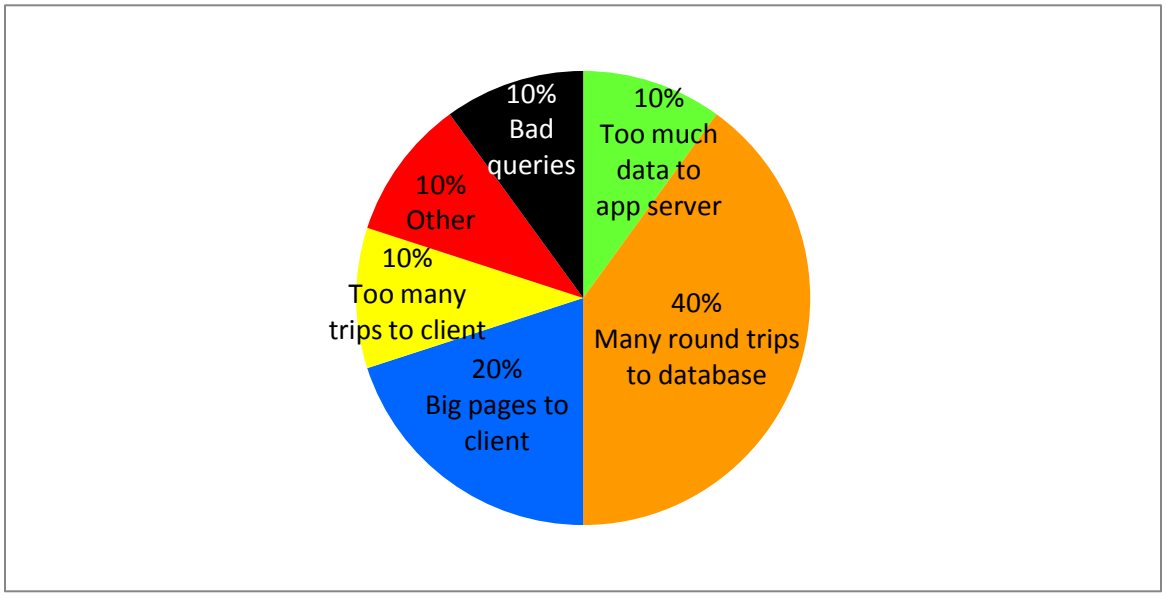

Figure 2: Sources of Performance Problems

Note that excessive round trips from the application server to the database are the leading causes of performance problems. The next most common cause of a performance bottleneck is the size of the page being sent to the client. Clearly a sound architecture should seriously address these two problem areas.

Applications requiring excessive round trips between the application server and the database make for wonderful, amusing anecdotes. At Fannie Mae some years ago, the month end batch routines were capable of processing one customer loan per minute. Each attribute read /write required a round trip between the application server and database. Given the 14,000,000 active consumer loans, this meant that the month end routine would require 26.5 years to complete. Needless to say, the routines had to be rewritten. At an application for the US Department of Defense, processing a single user data entry screen required 60,000 round trips between the application server and database. For a batch routine in the US Air Force Reserve recruiting system, Dulcian developers refactored a single Java routine (making thousands of round trips between the application server and database) that took 20 minutes to execute into a PL/SQL routine that executed in .2 seconds.

Large page sizes were an even bigger problem in the Ethiopian system. Discussions of the requirements with a Web Center expert from Oracle revealed that "Some of our pages are less than 1MB." Waiting 10 seconds for a single page to load was not an option.

The Solution

The solution was to create a framework with two main criteria:

1. A single round trip from the application server to the database for each UI operation
2. Reducing the amount of information sent to the client to the logical minimum

Minimizing rounds trips to the database meant moving everything into the database. Alternatively following the No-SQL approach would mean moving everything to the application server. The idea is to avoid the unnecessary context switches. Eventually, in order to achieve desirable performance, it was even necessary to move all UI logic to the database. Ultimately this evolved into the concept of an "ultra-thick" database, meaning that the entire application including data, logic, and UI layout were all stored in the database.

Minimizing page size sent to the client required a new way of thinking. Standard high band width pages are now measured in megabytes. Even basic HTML pages are often 40-100K in size. The finance system for Ethiopia required pages that were no more than 10K in size or the system would not support our band width requirements.

In order to minimize page sizes, it was necessary to ask the question: "What do we REALLY need to send?" To start with, a basic XML logical representation of a page was used:

```
<Page height = "200" ...>
 <Field height = "20" .../>
 <Field height = "20" .../>
 <Button label = "Save" .../>
</Page>
```

It was observed that with everything necessary for a page could be contained in a few kilobytes (rather than 40-100K). Therefore, this kind of logical description of the page was used along with a renderer on the client to create the actual page layout.

Using this approach, it was possible to reduce the UI to 10K or less. This could be cached on the client the first time that the page is loaded so that subsequent viewings of the page only required 1K or less data to be transmitted.

The realization that this strategy was what was needed led to some uncomfortable conclusions:

1) This architecture does not currently exist.
2) Industry standards would not be useful at all.
3) It is necessary to keep a complete copy of the UI state in the database.
4) A super smart "browser" is required.
5) The Application Server has a minimal role.
6) An ultra-thick database is needed.
7) Only minimal runtime logic can be sent to the client.

Given the other constraints of the project, the new system also needed to be:

1) Simple to learn/use
2) Productive
3) Able to produce functionally complete Web 2.0 pages
4) Rule-based and not dependent upon any specific architecture
5) UI tech stack-independent to run in thick client Java as well as browser-based applications.

Implementation

The core idea of the implementation was a new design pattern that we called the "Event-Action Framework" as illustrated in Figure 3.

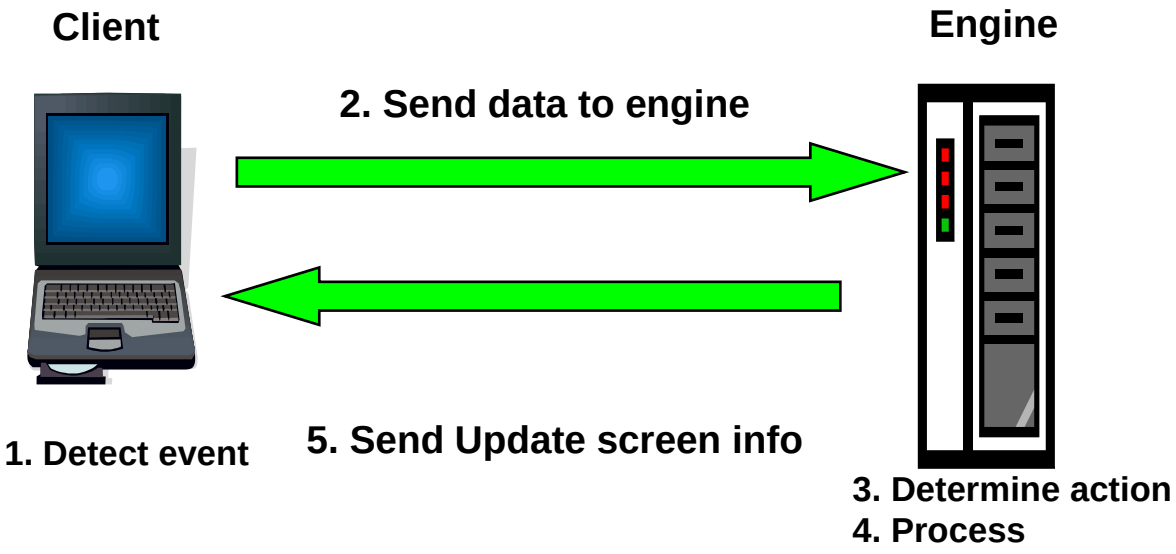

Figure 3: Event Action Framework

An event is detected in the client. This event along with any data that has been changed since the last event is sent to the server. The server accepts the changed data and processes the event. The engine determines how the UI must change and sends a set of change events back to the client. The client then interprets those change events and updates the screen. This is fundamentally different from how web applications typically work.

There are three components necessary in order to make this framework operate:

1) A client-side event detector/transmitter and an action interpreter
2) An engine to perform all of the work
3) A communication framework that allows the client and engine to communicate

The event detector/transmitter and action interpreter are client-side programs, which can either be native programs (C++ or Java) or browser extensions (e.g. a JavaScript library).

The engine requires four components:

1) A repository
2) A scripting language
3) A runtime engine
4) An IDE

The repository is a standard logical UI repository. A UML representation of the repository is show in Figure 4:

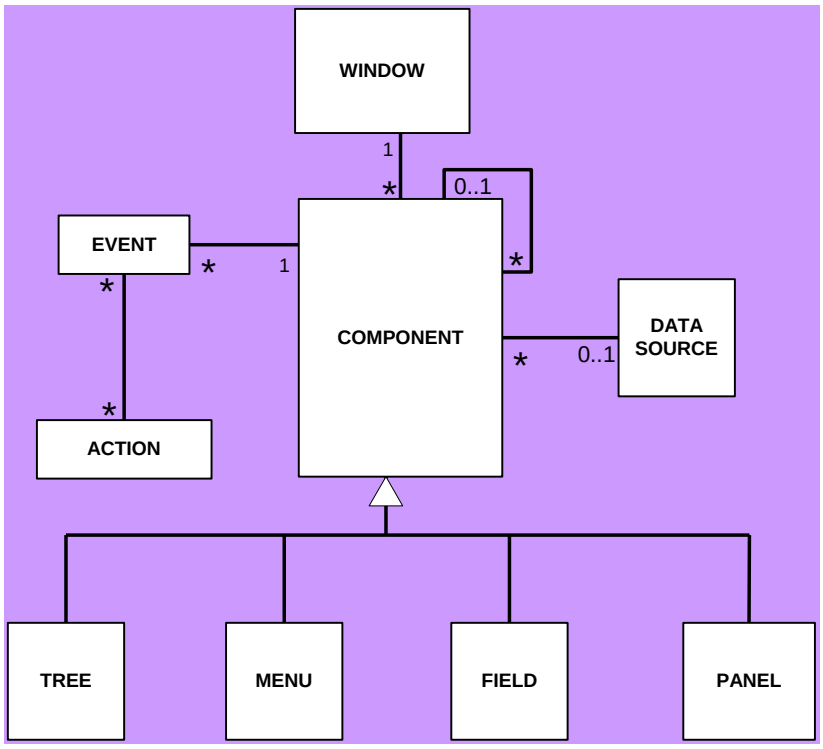

Figure 4: UI Repository

For the scripting language, PL/SQL was used because it was familiar to our developers. Several API packages were written to give developers access to the UI components. A sample code fragment is shown here:

```
procedure p_find is
     v_id number;
begin
  v_id:= api_component.getvaluenr('Demo01Panel.OID');
  api_datasource.setbindvar('Demo01.oid',v_id);

  api_datasource.executequery('Demo01');
end;
```

- "api_component.getvaluenr" retrieves the value from a UI component.
- "api_datasource.setbindvar" sets the bind variable of a data source.
- "api_datasource.executequery" executes the query of the data source.

The runtime engine is a combination of PL/SQL packages and database tables used to store runtime instances of UI components and data. The IDE is a customized repository manager written using the framework.

Conclusions

Using this approach, it was possible to create an easy-to-use UI development framework that satisfied the demands of the Ethiopian environment. The main achievements were:

1) A 90% reduction in network traffic over basic HTML-style applications
2) A very easy-to-use framework that could be easily and quickly taught to local developers. Developers only needed to learn SQL and PL/SQL.
3) A framework that supported high quality AJAX style UI development without the associated bandwidth impact.

About the Author

Dr. Paul Dorsey is the founder and president of Dulcian, Inc. an Oracle consulting firm specializing in business rules and web based application development. He is the chief architect of Dulcian's Business Rules Information Manager (BRIM®) tool. Paul is the co-author of seven Oracle Press books on Designer, Database Design, Developer, and JDeveloper, which have been translated into nine languages as well as the Wiley Press book *PL/SQL for Dummies*. Paul is an Oracle ACE Director. He is President Emeritus of NYOUG and the Associate Editor of the International Oracle User Group's SELECT Journal. In 2003, Dr. Dorsey was honored by ODTUG as volunteer of the year, in 2001 by IOUG as volunteer of the year and by Oracle as one of the six initial honorary Oracle 9*i* Certified Masters. Paul is also the founder and Chairperson of the ODTUG Symposium, currently in its eleventh year. Dr. Dorsey's submission of a Survey Generator built to collect data for The Preeclampsia Foundation was the winner of the 2007 Oracle Fusion Middleware Developer Challenge and Oracle selected him as the 2007 PL/SQL Developer of the Year. Paul can be contacted at paul_dorsey@dulcian.com