Bubba's CloudBooks

A Cloudscape ePerformance Test

Cloudscape

Introduction

This paper describes the architecture of Bubba’s CloudBooks, an eBusiness test application that shows Cloudscape™ running embedded in a Web application. As a 100% Pure Java™ relational database management system, Cloudscape is perfectly designed for a Web application based on Java Server Pages™.

What Is Bubba’s CloudBooks?

Bubba’s CloudBooks is an online bookstore. It does not compete with the big book stores; instead, it focuses on carrying the best 10,000 books on clouds. By today’s standards, Bubba’s bookstore is not huge, but it does have over three million loyal customers who have ordered over seven million books. Bubba’s database has grown to more than six gigabytes.

Bubba’s CloudBooks was created to test the scalability of Cloudscape in a real-life Web application. With its small footprint and full object-relational database functionality, Cloudscape was designed from the start to be a multi-user small- to mid-size database management system.

Test Application Goal

The goal in creating Bubba’s CloudBooks was to build a real eBusiness environment to test database performance, so Bubba’s was developed with all the appropriate database transactions and full customer user interface of an eBusiness. Some non-database functions were left out and some non-real life features were added to make Web testing easier and to shorten development time. For example, when you enter credit card information, the application only checks the format of the date and, as you might guess, does no credit card authentication. Also, the shopping cart always contains at least one item, which makes testing easier if, for example, you navigate from Home to Shopping Cart to Checkout. These features do not decrease the amount of work for the database; in fact, they require additional transactions in most cases.

Technical Details

Database Schema

The database schema contains nine tables:

Table	Description
Address	Customer address

Table	Description
Author	Book authors
Cart_Items	Items placed in the shopping cart
Country	Country name and ID
Customer	Customer information
Item	Book titles and descriptions
Order_Line	Line items for the orders
Orders	Order information
Shopping_Cart	Shopping cart information

Interaction Mix

Bubba's CloudBooks incorporates the most common database transactions required for an online store. Database read operations include looking up the details of a book and displaying the latest books on the market. Database alter operations include adding items to the shopping cart and placing an order. During a test run, the database interactions include the following:

- Single row reads, updates, inserts, and deletes
- Multi-row inserts, updates, and deletes
- Multi-table aggregations

A test run consists of approximately 85% browsing and 15% purchasing. Browsing includes adding items to and removing items from the shopping cart.

Application Architecture

Bubba's Web site is composed of Java Server Pages (JSP), which contain the presentation logic, and Java Beans™, which contain the business logic. The application was compiled using the Sun® Java™ Development Kit 1.2.2_05a.

Accessing the Database

To begin our look at this architecture, we will examine how the item detail page is implemented. The item detail interaction uses item_detail.jsp, GetDataDetail.java, and ProductDetail.java:

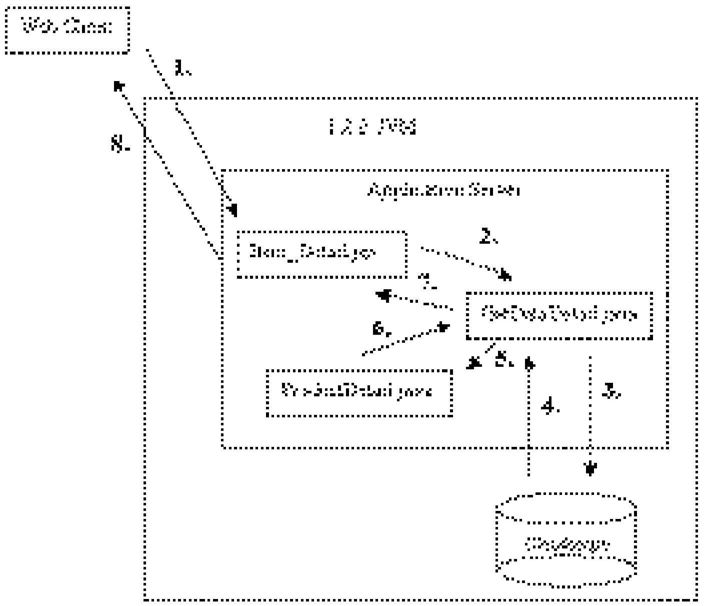
- **Item_detail.jsp (JSP):** The presentation component, responsible for processing any posted parameters, collecting session objects, calling the Bean to get data, and generating the HTML that is returned to the client.
- **GetDataDetail.java (Java Bean):** Contains a method that accesses the database to collect the item information and populates a ProductDetail object.

- **ProductDetail.java (Data Object):** An object with fields that contain details about the book, such as author, title, and number of pages. The fields in the ProductDetail object contain a subset of information from the author and item tables provided by the getProductDetail(int i_id) method of GetDataDetail.

  The getProductDetail method has the form:

  ```
  public ProductDetail getProductDetail(int i_id) {
     //Select the required fields from the database
     //Based on an i_id (item ID)

     return(Populated ProductDetail object);
  }
  ```

The following figure shows the processing for the item detail page:



1 The Web browser makes a request to the Web server (http://server/item_detail.jsp?I_ID=567).

2 Item_Detail.jsp parses the i_id from the URL parameter and passes it to the getProductDetail(i_id) method of GetDataDetail.

3 The getProductDetail method makes a JDBC call to the Cloudscape database to get the information:

```
SELECT I_title, a_lname …
FROM Item, Author
WHERE I_id = ?
AND a_id = I_a_id;
```

4 The result set is returned to getProductDetail( ).

5 The ProductDetail constructor is called with the data from the database.

6 A populated ProductDetail object is created.

7 The getProductDetail method returns the ProductDetail object to item_detail.jsp.

8 The HTML for the Web page is generated and returned to the Web browser to be displayed.

This methodology is used throughout Bubba's Web site to retrieve, update, insert, and display data.

However, moving data to and from the database is only part of the application. We will also look at how to use some features of Java to create the best performing application and how to gain application performance using the servlet session API and static class variables.

We will now take a look at how Bubba's CloudBooks implements the shopping cart.

The Shopping Cart

For our shopping cart, we have two basic design parameters:

- The shopping cart must use Java features to improve performance.
- The shopping cart must be persistent. (The cart for a known customer must be able to be retrieved when that customer returns to the store.)

The cart requires one JSP (cart.jsp), a Bean (CartData.java) with all of the business logic, and an object container (Cart.java).

For persistence, we use two database tables in the Cloudscape database:

Table	Description
Shopping_Cart	Contains all the information about the cart and its owner (cart_id, cust_id, etc).
Cart_Items	Contains the data for the individual line items.

CartData.java contains methods for creating a shopping cart, adding an item to a cart, and so on. Every operation that adds items to the cart updates the database and a shopping cart Vector. The information for each cart item is stored in a Cart object. A list of Cart objects is stored in a Vector. This Vector is stored in memory on the server in a session object and can be referenced by a JSP.

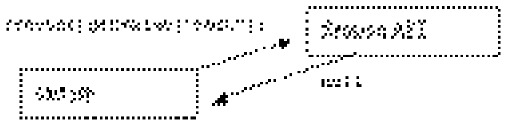
To see how session objects are managed, we will look at steps for creating a shopping cart for a new user.

Shopping Cart Processing

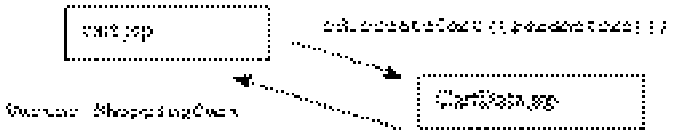
The processing of the shopping cart proceeds as follows:

1 The Web client starts at home and requests http://server/cart.jsp.

2 Cart.jsp checks the session and determines that a shopping cart does not exist for this customer (i. e., the Vector does not exist in the session).



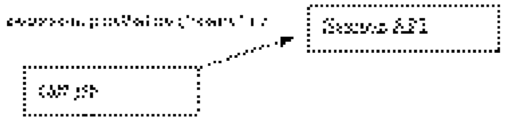
3 Cart.jsp calls the cart.createCart(String shopping_id) method, and the shopping cart Vector is populated with a random item (an ease-of-testing feature).



4 Cart.jsp displays the results by printing the contents of the shopping cart Vector.

```
for(int item_count = 0;item_count < shoppingCart.size();item_count++) {
   Cart cartItem = (Cart) shoppingCart.get(item_count);

   out.println("<tr><td width=\"10%\"></td><td "+
      "ALIGN=\"right\"  VALIGN=TOP width=\"5%\"> "+
      "<INPUT NAME=\"qty1\" SIZE=4 TYPE=\"TEXT\" "+
      "value=\""+itemQty+"\"></td>");

   out.println("<td VALIGN=TOP width=\"85%\" "+
      "colspan=\"6\">Title:<i>"+
      cartItem.getTitle()+"</i>");

   out.println("<br>SRP. $"+itemSrp+", <b><font "+
      "color=\"#AA0000\">Your Price: "+
      "$"+itemCost+"</font></b></td></tr>");

} //end for
```

5 The Shopping cart Vector is placed in the session for this user.



By maintaining the shopping cart in the session, we can:

- Easily display the contents of the shopping cart Vector.
- Display the cart if the contents do not change.
- Use the Vector to process changes before they are written to the database (for example, to change the quantity of an item already in the cart).

The shopping_id in the shopping_cart and cart_item tables is the servlet session id (session.getId( );). This way, we do not need to generate IDs and can let the application server manage unique ID generation.

Session objects stay in the session cache on the server. As a result, putting information into the session does not add to network traffic because only a pointer (cookie) is copied to the client to identify the client for a given session.

We have looked at accessing the database to retrieve data and using the servlet session to manage our shopping cart. We will now take a look at using object persistence to solve a common problem—generating unique IDs.

Generating Unique IDs

When adding rows to a table that has a unique key, you must generate a new key value. Two common ways to create unique IDs when inserting rows are:

- Use an incrementing column type. Some databases, including Cloudscape 3.6, have an auto-incrementing column type that creates unique IDs automatically.
- Use a “Select max(c_id) from customer” statement and add 1 to it.

However, not all databases include the incrementing column type, and with the “max” option, you must lock the table to maintain uniqueness. So Bubba decided to use static class variables to generate unique IDs quickly.

To generate IDs quickly, we need to create the MaxId.java class that will manage the max ID variables.

```
package jdbc.CloudDemo;

import java.util.*;
import java.sql.*;
import java.math.*;

/*****************************************************************/
/*  This class returns a unique ID for the following Column:     */
/*     c_id                                                      */
/*****************************************************************/

public class MaxId{
   //We create a variable to hold the latest c_id
   protected static Integer custId;

   public MaxId()
      throws SQLException
   {
      Connection connection = null;
      PreparedStatement pstmt = null;
      ResultSet result = null;

      If (custId == null) {
         try {
            //Get Database Connection
```

```
                DemoConnect newConn = new DemoConnect();
                connection = newConn.getConnection();

                //Customer ********************
                //We do a max() but we only have to do it
                //once each time the application server is
                //started
                pstmt = connection.prepareStatement("SELECT "+
                                    "max(c_id) FROM customer");
                pstmt.execute();

                result = pstmt.getResultSet();
                result.next();
                custId = new Integer(result.getInt(1));
            }
            finally {
                result.close();
                pstmt.close();
                connection.close();
            }
        } //End if values are empty
    }

    public Integer getNextCustomerId()
    {
        synchronized (custId) {
            custId = new Integer(custId.intValue() + 1 );
            return(custId);
        }
    }
}
```

When the MaxId class is first instantiated, we need to find the current maximum ID in the customer table. Therefore, in the MaxId( ) constructor, we check to see if custId has been set. If custId has not been set, we query the customer table to get the current maximum customer ID. This process makes the class self managing—if the application server is restarted, it will be re-initialized with the most current values the first time MaxId( ) is used.

Then we create the getNextCustomerId( ) access method to get the next new customer ID. In this method, we take the current custId, add one, and return the result.

Since this static variable will be shared, we need to make this code thread-safe. To keep multiple threads from changing the custId variable at the same time, we wrap it in a synchronized block.

This MaxId class gives us an efficient way to generate unique customer IDs. Bubba uses this same methodology for all unique ID generation.

Performance

Now that we have seen how Bubba's CloudBooks is designed, we can take a look at the results of the performance tests.

Test Methodology

To test the performance of Bubba's online bookstore, we needed to generate hundreds of simulated Web clients. For this task, we used LoadRunner® from Mercury Interactive. The vuser (Virtual User) script starts at the home page and randomly select options on each page until a specified time-out. The users are configured to run 1000 iterations to ensure a constant load on the server.

Test Configuration

Server :

- Sun SunOS™ 5.7
- 4 x 295 MHz SPARC™ processors
- 512 MB memory

Client:

- Client load was generated from a mix of six PCs running Windows NT®.
- Each user is a returning customer (to simplify the test environment and take some load off the clients).
- The client think-time is a random time between 2000 and 8000 milliseconds.

Server Response Measurements:

- A test run was determined to be successful if it maintained a less than one second average response time for all interactions during a run of 180 seconds. There was a 5–10 minute warm-up before any measurements were taken.
- If there was a peak of more than 10 seconds for an interaction, it was considered an anomaly. Measurements which contained such a peak were not included.

Application Software:

- Database: Cloudscape 3.5
- Application/Web Server : Enhydra™ 3.0.1
- Java Virtual Machine: Sun 1.3 Release Candidate

OS:

- Installed recommended Java 1.3 patches
- Changed /etc/system settings to:

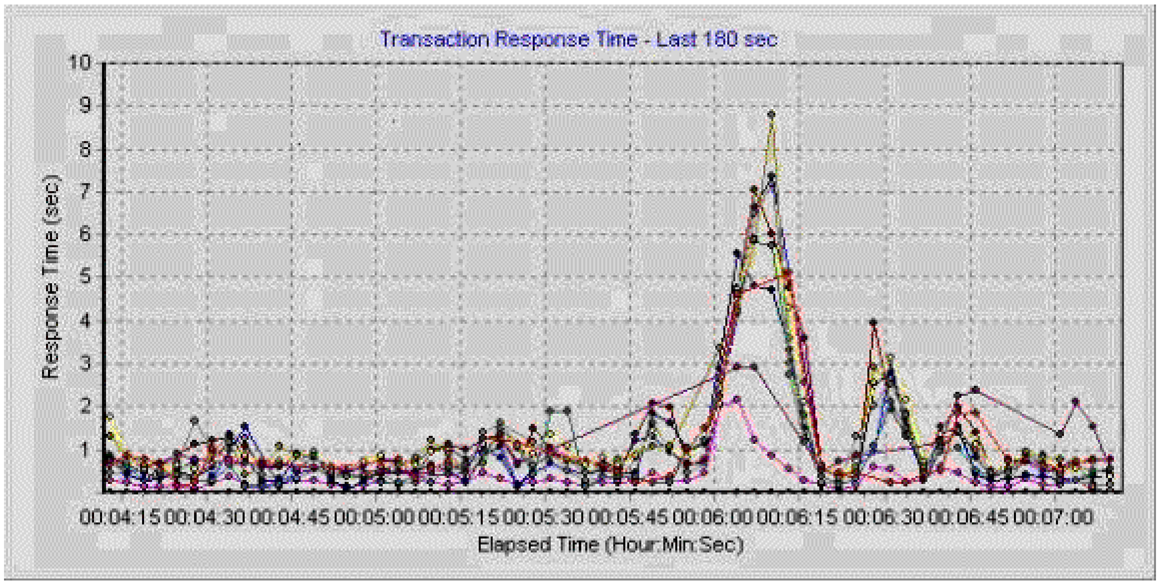
```
set rlim_fd_max = 4096
set rlim_fd_cur = 1024
maxusers=1000
set lotsfree=0x40000
```

Java:

- Native Threads
- 128 & 256 MB of Java Heap

Test Results

The graph shows the response time for each Web interaction. Response timing starts when a page is requested and ends when the last character has been received by the client. Each line on the graph represents a different type of interaction (home, shopping cart, item detail, and so on).



During this period, 300 users ran a combined 46.3 Web interactions per second. Each Web interaction submitted multiple database interactions, which translated to 232 database interactions per second.

Note: The ~8 second peak you see in the graph was caused by full garbage collection.

Resource Links

Mercury Interactive:

http://www-heva.mercuryinteractive.com/

Author:

Scott Fadden
Cloudscape Performance