Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

VOL. 18, No. 1 · FEBRUARY 2004 $15

Special NoCOUG Training Day

Don't misss this rare opportunity!
See page 27.

The Case of the Missing Kanji

Read about the database adventures of Brian Hitchcock.

See page 13.

A Conversation with Gaja Krishna Vaidyanatha

Author of Oracle Performance Tuning 101

See page 5.

BAARF – It's Not What You Think!

The scoop on why some people are saying, "Enough is enough."

See page 10.

Your Career Takes Off with NoCOUG in 2004!

We have some fantastic things planned in 2004 that will help your career take off! With NoCOUG, you always have the opportunity to network, learn, and share information with your peers. There are great conferences filled with technical sessions and roundtable discussions. Plus, there are plenty of volunteer opportunities to get involved in: speaking, serving on the board of directors, writing for the *NoCOUG Journal*, volunteering at the conferences. Just contact us at board@nocoug.org to get involved in 2004.

Mark your calendars now!

- ➤ Winter Conference on February 19, Oracle Conference Center, Redwood Shores
- ➤ Training Day on April 8, Dublin Library, Dublin
- ➤ Spring Conference on May 13, Lockheed Martin, Sunnyvale
- ➤ Summer Conference on August 19, Chevron/Texaco in San Ramon
- ➤ Fall Conference: TBD

Why Should You Make a Presentation at a NoCOUG Conference? See page 24.

The Case of the Missing Kanji

By Brian Hitchcock, Sun Microsystems

How did all this get started? I'll tell you how. The interviewer wore a very loud Hawaiian shirt—that's how it all started. I should have been more careful. I assumed this meant exotic job locations. It meant working out by the salt ponds, where watching salt water evaporate is a large-scale industrial concern.

I knew Sybase as well as Oracle and that meant I was the perfect "volunteer" for the migration. It all sounded so simple. They couldn't find anyone else with this strange set of qualifications. I had been chosen. I used Oracle Migration Workbench, which worked very smoothly. The migration went well, perhaps too well.

And then it happened. Of all the DBA joints in all the corporate world, she walked into mine. Christine said little. "Brian, the Kanji data are missing." I knew I was in trouble. Up to that point, no one had mentioned the minor detail that there was multi-byte Japanese Kanji in this innocent looking single-byte database.

The "expert" DBAs at corporate analyzed the situation in the predictable way. "Brian, a single-byte database can't support multi-byte Kanji." And that was that. I was stuck, in the middle of nowhere, between Christine's need for her Kanji, and the experts' declaration that the Kanji had never existed.

This left only one small detail. How to get the Kanji back?

Before the upgrade to Oracle, Christine's application had been working for years without a problem. The application retrieved data from the single-byte Sybase database and sent the data to the user's browser where the Kanji were displayed. Everyone was happy. When I moved the data from Sybase to Oracle, I had no reason to worry. I created the Oracle database using the WE8ISO8859P1 character set to match the ISO1 character set of the existing Sybase database.

The way the application worked is shown in Figure 1. The source system was sending Japanese Kanji characters into the Sybase application database. The application would retrieve data from the Sybase database to generate HTML that was sent to the end user's browser. Notice that in the lower left of the figure, I have included a single Kanji character. It would turn out later that the Japanese Kanji data for this application were from the EUC-JP character set. I have included the byte code for this specific Kanji character in the EUC-JP character set, which in hexadecimal is B0A1. Notice that this byte code for this specific Kanji character was stored as B0A1 in the source system and was sent to the Sybase database as B0A1. The application retrieved this double-byte byte code and sent it to the end user's browser. The Netscape browser examined this byte code and then displayed the proper Kanji character. Notice that the Netscape browser was configured to use the character set called Japanese auto detect (under the View menu, Character Set). The Netscape browser looked at each byte to be displayed, and if the eighth bit was set to 1, it assumed that this was the first byte of a double-byte byte code, a byte code for a Japanese character. By following this specific byte code for this specific Kanji character as it moves through the application, we can see how the problem developed.

Brian Hitchcock

I moved the Sybase data to the Oracle database using Oracle Migration Workbench, which generated SQL*Loader scripts for each table in the Sybase database. I then executed these scripts. Everything ran well with no errors. I created the Oracle database using the WE8ISO8859P1 character set to match the ISO1 character set of the existing Sybase database. The SQL*Loader scripts were executed and I didn't worry about character sets, which meant the client environment where SQL*Loader was executing used the default character set, which for Oracle is US7ASCII. The US7ASCII character set uses a single byte to represent each character. Within each byte, only seven bits are used to represent the

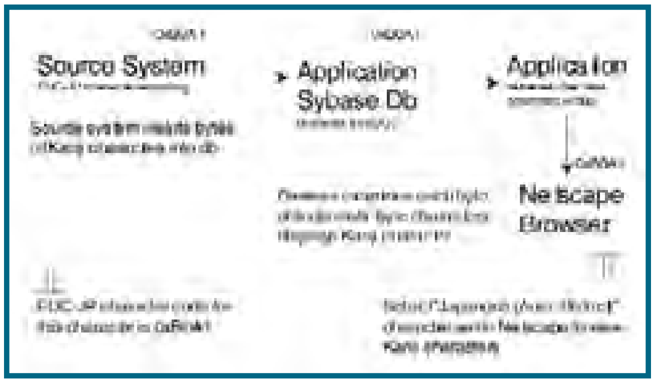

Figure 1. Application Before Upgrade to Oracle

binary code for each character. The highest bit, the eighth bit, is set to zero for every character in the character set. Another way to look at this is that the decimal value for all the US7ASCII characters is in the range 0 to 127.

When you use any Oracle utility, such as SQL*Plus and SQL*Loader, the environment parameter NLS_LANG determines the character set used by the client utility program. In this case, since I didn't set NLS_LANG, it defaulted to US7ASCII.

However, it turns out that the Sybase database had also been created with the default character set, which for Sybase is called ISO1. Notice that different vendors have very different names for character sets that may turn out to be the same. In this case, the Sybase character set called ISO1 is the same as the Oracle character set called WE8ISO8859P1. This character set uses eight bits to represent each character. The Oracle database where I was loading the data had been created with the WE8ISO8859P1 (Oracle version of the 8859-1) character set. This was done simply to match the existing production databases.

Notice that the existing Sybase database had the byte code for our Kanji character of B0A1. This byte code was then moved into the flat file produced by the Sybase BCP utility. This flat file was loaded by the Oracle utility SQL*Loader into the Oracle database and the byte code changed from B0A1 to 3021. Once this byte code was stored in the Oracle database, the application retrieved the two bytes of this byte code, generated the HTML, and the byte code 3021 was displayed in the Netscape browser as two characters (two ASCII characters) zero followed by an exclamation point. Clearly things were not correct.

What was happening was that ASCII data was being moved into a database that was created with the WE8ISO8859P1 character set. Therefore, when I executed Oracle SQL*Loader with NLS_LANG defaulting to US7ASCII, the Oracle software looked at the data in the SQL*Loader datafile and treated each byte that was loaded as if it came from the US7ASCII character set. This is exactly what Oracle was being told to do. It also meant that Oracle was setting the eighth bit of each byte to zero before loading it into the Oracle database. There were no error messages generated. As far as Oracle was concerned, everything was fine. Any byte in the Sybase database that had the eighth bit set to 1 was loaded into the Oracle database with the eighth bit set to zero.

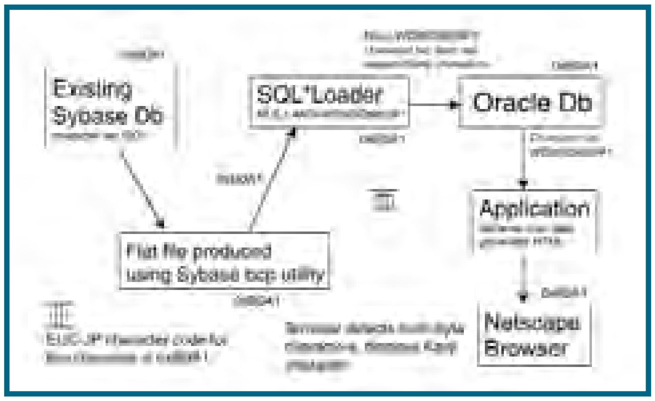
Existing Sybase Db
SQL*Loader
Oracle Db
Application
Flat file produced using Sybase bcp utility
Netscape Browser
EUC-JP character code for

Figure 2. Fix–Sybase Data to Oracle

The bit-by-bit details of what was happening are tedious, but worth following. Starting with the correct two-byte byte code in hexadecimal for our selected Kanji character (B0A1), the binary digit equivalent of B0A1 is 1011 0000 1010 0001. When we change the eighth or highest order bit to zero for each of the two bytes, the resulting binary digits are 0011 0000 0010 0001, and the hexadecimal equivalent of these modified binary digits is 3021. The byte code 3021 corresponds with the character zero followed by an exclamation point.

This wouldn't have been an issue, except for the fact that we had multi-byte Kanji stored in the single-byte Sybase database. Each Kanji character is represented (in this specific case of the EUC-JP character set) by two bytes. The way the Kanji characters are encoded into two bytes causes the eighth bit of both bytes to be set to 1. (Each byte has a decimal value of 128 to 255.) As far as Sybase was concerned, these weren't Kanji characters at all, they were simply two bytes in a single-byte database. Somehow, these two bytes were retrieved by the application and sent to the user's browser and the Kanji character was displayed. Note that the Sybase character set ISO1 uses all eight bits for each character, so the bytes stored in the Sybase database were both correct ISO1 characters. The trick was that when these two bytes were sent to the user's browser, they were somehow combined and instead of two ISO1 characters (a-z, A-Z and some special characters for European languages) a single Kanji character was displayed.

When these two bytes were loaded into the US7ASCII Oracle database, the eighth bit of both bytes was set to zero. The two bytes of the Kanji character now appeared to be two characters from the US7ASCII character set. The user's browser was not able to display a Kanji character from these two seven-bit bytes. This is how the Kanji were lost.

Now that I could explain what had happened to the Kanji, the immediate concern was how to fix it. The answer comes from understanding how Oracle treats character data when moving between SQL*Loader and the database.

Oracle looks at NLS_LANG in the client environment where SQL*Loader is executed, and compares that with the character set of the database. If the character set and NLS_LANG both specify the same character set, Oracle doesn't do anything to the bytes, it simply loads them into the database. Note that it doesn't matter if the bytes represent correct characters in the specified character set. Oracle doesn't compare the bytes to the character set of the database or the client, it simply loads the bytes. In this case, SQL*Loader was using US7ASCII while the database had been created with the WE8ISO8859P1 character set. Therefore, the fix was simply to modify the client environment so that the ISO1 character set was specified. For Oracle, the equivalent character set to Sybase ISO1 is WE8ISO8859P1. Once this change was made, Oracle detected the same character set at the client and at the database and no changes were made to any of the bytes.

Figure 2 shows the situation after the fix was made. In the client environment where the SQL*Loader utility was executed, NLS_LANG was set to WE8ISO8859P1. Now that the character sets were the same between the SQL*Loader client environment and the Oracle database that SQL*Loader was

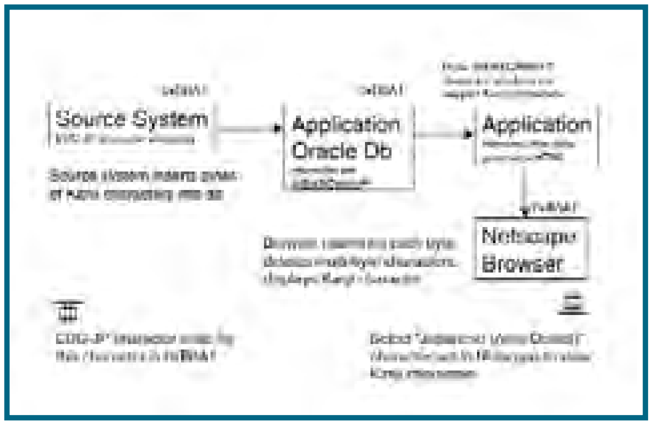

Figure 3. Current Oracle Production

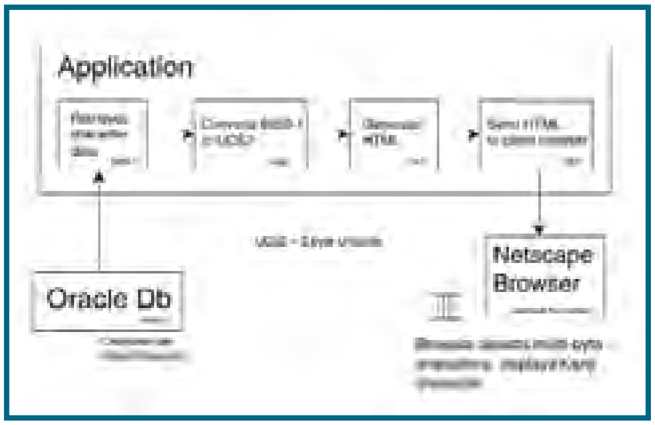

Figure 4. Existing Application

connected to, Oracle made no attempt to examine or convert the bytes as they moved from the flat file through SQL*Loader into the Oracle database. The byte code of our selected Kanji character (B0A1) moves through the system without being changed, and therefore was loaded into the Oracle database as B0A1. The application could then retrieve this byte code and send it to the Netscape browser where it was displayed as the correct Kanji character.

At this point, the bytes that represented the Kanji characters were loaded into the Oracle database. The bytes of each Kanji character that had been in the Sybase database were now in the Oracle database. Remember that both the Sybase ISO1 and the Oracle WE8ISO8859P1 character sets are single-byte character sets. Neither of these character sets supports (officially) multi-byte character sets. On the other hand, since the application was able to retrieve Kanji characters from both databases, it was clear that something else was happening.

After the migration from Sybase to Oracle, the production application was working and everyone was happy. The fact that the database character set didn't support Kanji (multi-byte character data) was not a concern to anyone, except for someone out by the salt ponds, but no one listened to him anyway.

Figure 3 shows the production system after the Sybase data had been successfully loaded into the new Oracle database. The byte codes for individual Kanji characters were moving through the entire application and were correctly displayed in the end-user's Netscape browser.

As exciting as it is to watch the salt brine evaporate, leaving salt behind, I was curious how the existing application was able to do what the experts said couldn't be done. We were in production with a system that didn't officially support Kanji, but they were being stored and displayed on a daily basis. Since I wasn't an expert, this bothered me.

To figure out what was going on required examining all the pieces of the application. The multi-byte Kanji characters were inserted into the Oracle database by another application. Further proof that Oracle (and Sybase) doesn't examine the bytes of character data as they move into and out of the database. The application code itself is Java. The Java code uses JDBC to retrieve the character data from the Oracle database. Java converts the character data to UCS2 (Unicode, two bytes per character) and generates HTML that is sent to the user's browser. The Netscape browser then examines the bytes sent to it. You can choose a character set within the Netscape browser, and for this application, the user always uses the "Japanese (Auto-Detect)" character set. This means that the browser examines the bytes of the character data. If it "detects" bytes where the eighth bit is not -zero, then it assumes that these bytes are part of a multi-byte character. Since the browser is "detecting" Japanese characters, it assumes that any multi-byte characters are Japanese, and displays them as Kanji. Also note that while Netscape was able to "detect" Japanese multi-byte character data, this only worked because all the multi-byte character data was Japanese. If there had been, for example, Chinese character data, which would also be multi-byte, Netscape wouldn't be able to "detect" Chinese versus Japanese character data and the application would not have worked. Figure 4 shows all the pieces of the application, and the various character sets that were being used by each piece. Notice that each component of the overall application was either configured for a specific character set or was defaulting to a specific character set.

I was also curious about the source data itself. What exactly was the character set used when the original Kanji character data was inserted into the database? And how could I determine this? I wanted to know what exactly had happened to get us into this mess. I looked at the actual bytes for a sample of the data in the database. I had to learn how to convert between the many different character sets that support Japanese Kanji characters. I also looked at the actual Kanji characters displayed by the application for the sample data I was examining. I then found the Kanji characters in a Japanese character dictionary, which offers the row-cell values for each Kanji character. I then converted the row-cell values into the equivalent bytes for several character sets that support Kanji and compared them with the bytes of my sample data. I then verified that the source data came from the EUC-JP character set. Note that in order to do this, someone had to go through all these steps. There are no Oracle utilities that would help me.

After this, I could see what had happened. The original Kanji characters were encoded in the multi-byte EUC-JP character set. The bytes of the Kanji data were inserted into the Oracle database (as they were before in the Sybase database). The application retrieved these bytes, and the client browser looked at the bytes and detected multi-byte data.

Perhaps I was done? Perhaps Christine would be content and I could return to the salt ponds? No. At the corporate level (where the experts live) the decision had been made that this application, as well as many others, should be converted to support multiple languages. The application was to be “internationalized.” The standard way to do this is to convert the Oracle database to the UTF8 character set which has unique bytecodes for the characters of many, if not all, the world’s languages. The UTF8 character set is Oracle’s implementation of the UTF-8 encoding scheme for the Unicode character set. A good idea, but this was coming from the same experts that said there couldn’t be any multi-byte data in the existing single-byte databases.

The standard way to convert an Oracle database from one character set to another is simple. I would export the existing Oracle database (character set WE8ISO8859P1) and import into another database that had been created using the Oracle UTF8 character set. The experts all agreed that this was all that would be needed.

After this process was complete the application was tested and it was able to retrieve and display the existing Kanji data. Everyone was happy. Everyone, except for that someone out by the salt ponds.

Since I had been deceived before (the original Sybase database didn’t contain any multi-byte data, remember?), I decided to look at what the export/import process would do to the bytes of a Kanji character.

I needed to examine the process that is used to encode a character in the UTF8 character set. I had to learn that the term “UTF8” means many things. UTF-8 is an international standard for encoding character sets. This means you can create any set of characters you like, and use the UTF-8 encoding method to encode your set of characters into bytes for storage in a computer. This means that the term UTF-8 doesn’t refer to any specific character set. At the same time, the Oracle 8i Unicode character set is also called UTF8 and refers to a very specific set of characters.

The UTF-8 encoding process takes the Unicode code point (the hexadecimal number of the character) and encodes that value into one, two, or three bytes. For ASCII characters, UTF8 uses one byte. Two bytes are used to encode the Unicode characters up to U+00FF, and three bytes are used for Unicode values above that. UTF8 uses three bytes for

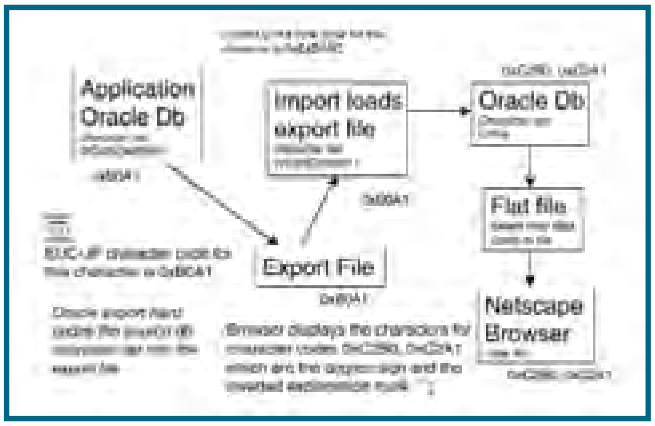

Figure 5. Conversion Issue

Asian characters. Note that UTF8 doesn’t use more bytes for the ASCII data. Converting to UTF8 character set will not increase the size of the ASCII character data. The bit-by-bit process of converting the EUC-JP byte code of our selected Kanji character to the new UTF8 byte code for the same character is as follows: We start with the EUC-JP byte code (two bytes), which is B0A1. The Unicode code point (byte code) for this same Kanji character is 4E9C. You can review various Oracle Metalink documents for the details of this conversion process. The Unicode byte code of 4E9C requires three bytes in the UTF8 encoding process. The binary digits for 4E9C are 0100 1110 1001 1100, and when encoded into UTF8 this becomes 11100100 10111010 10011100 or E4BA9C. For the same Kanji character, the bytes for EUC-JP are B0A1 and in UTF8 they are E4BA9C. Note that the byte code was two bytes in EUC-JP, and it became three bytes in new UTF8.

There are multiple character sets for Japanese and other Asian languages. The same Kanji character has a different byte code in each of these character sets. I had to figure this out to know what the bytes should be in the UTF8 database. Details of this process are covered later on. I used this to compare with the bytes of the original Kanji character data to determine that they came from the EUC-JP character set.

When the existing data was exported from the WE8ISO8859P1 database, the export utility assumed that each single byte exported represented a single character. As each byte was inserted into the UTF8 database, Oracle converted each single byte into the equivalent UTF8 byte or bytes. This means that the two bytes for a Kanji character in the existing database were converted one byte at a time. Each byte became two bytes in the UTF8 database. The correctly encoded UTF8 bytes for the same Kanji character consist of three bytes. I now had evidence that simply exporting from WE and importing into UTF8 did not produce the correct results. The bytes in the UTF8 database were not the correct bytes for the Kanji in the UTF8 character set.

I had to go through the conversion manually to see what was happening as the import process moved the character data into the UTF8 database. Oracle took each single byte that was exported from the WE8ISO8859P1 database, determined the Unicode character and then applied the UTF8 encoding process. This is very different from what should have been done. The two bytes of the Kanji character in the WE8ISO8859P1 database should be exported together. The Unicode character for these two bytes should be determined. Then, this Unicode character should be put through the UTF8 encoding process to generate the correct UTF8 bytes (three bytes) for the Kanji character.

Manually generating the UTF8 bytes for each single byte of the existing two bytes of a single Kanji character demonstrates exactly what happened. Starting with the two bytes of our EUC-JP Kanji character, B0A1, the import process took each of the two bytes one at a time. The first byte, B0 was converted to UTF8, which results in C2B0, and the second byte A1 becomes C2A1. The result of importing into the UTF8 database was to change B0A1 into four bytes C2B0C2A1. Notice that the byte code for our selected Kanji character, B0A1, which is two bytes, became four bytes after the export file was imported into the Oracle database. This process is shown in Figure 5.

This explains how the Kanji data would be corrupted (lost) in the UTF8 database. The two bytes of the Kanji character in the WE8ISO8859P1 database were converted by export/import into four bytes in the UTF8 database. These four bytes represented two characters in the UTF8 database. The correct (manually generated) UTF8 encoding for the same Kanji character has three bytes E4BA9C.

What had happened? Oracle did exactly what it was told to do. The export utility looked at the database it was exporting from and saw the single-byte character set WE8ISO8859P1. Oracle correctly determined that each byte coming from this database represented a single character. The import utility inserted each byte into the UTF8 database. Oracle compared the character set of the export file to the character set of the UTF8 database and knew that character set conversion was required. Oracle then took each single byte from the export file and converted that single byte to the correct UTF8 set of bytes. But, this wasn't the correct set of UTF8 bytes for a Kanji character.

But, Christine was happy to report that the application was working just fine with the existing data that had been exported from the WE8ISO8859P1 database and imported into the new UTF8 database.

I had to tell her that maybe not today, maybe not tomorrow, but someday, during the rest of the life of this application she'd regret it.

Christine was not amused. Why couldn't it work? Why not for us? Why worry about the future? Couldn't we just be friends?

Because I'm not an expert, I had compared the correct UTF-8 bytes for a single Kanji character with the bytes that resulted from export/import for the same character. The bytes were not the same. But so what? Who cares?

The problem comes in the future. Up to this point Christine and I had only been concerned with the existing data. After the existing data was exported and imported into the UTF8 database, the application worked fine. But, what would happen when, in the future, some process inserted the correct UTF8 bytes for this same Kanji character into the UTF8 database? At that point, the UTF8 database would contain, for the same Kanji character, four bytes from existing data, and three bytes from newly inserted data. How would the application know which was which? Further, recall that the existing application works only because the Netscape browser is configured to use the Japanese (Auto-Detect) character set. Is the application really "UT-F8" if it requires using a specific Japanese character set in the browser? Is the application really "UTF8" if it has both correctly and incorrectly encoded bytes for the same character?

How could the existing application work with what I believed was corrupted (four bytes versus three bytes) Kanji character data? I had to review how the application worked to see the answer. The application uses Java and JDBC to retrieve the bytes from the UTF8 database. Java then converts these bytes to the Unicode character, generates HTML, and sends the HTML to the user's browser as shown in Figure 6.

We now had four bytes in the UTF8 database that represented the UTF8 conversion of each of two bytes of the original Kanji character data in the EUC-JP character set. When Java converted these four bytes into Unicode, it got the correct European characters for each of the pairs of bytes. Note that these two characters are exactly the same characters that were inserted into the original Oracle WE8ISO8859P1 database. The application sent these two characters in HTML to the browser. The browser then detected these eight-bit bytes and displayed the correct Kanji character!

What had happened was that the bytes of each Kanji character were converted to UTF8 (one byte at a time) when they were inserted into the UTF8 database. At this point, the original two bytes of a single Kanji had been encoded to UTF8. Java then reversed this process, which simply returned the original two bytes. The browser then detected the two bytes and displayed the correct Kanji character.

Overall, the export/import to the UTF8 database hadn't really changed anything. The application still delivered the same two bytes to the browser that it did before UTF8 had entered the process.

I decided to test the existing application by inserting into the UTF8 database the three bytes that represent the correctly encoded UTF-8 for a specific Kanji character. The application didn't work!

The existing application was able to work with the existing data that had been exported from WE8ISO8859P1 and imported into the UTF8 database. But, the existing application didn't work when it tried to work with correct UTF8 data. I even tried changing the browser character set to UTF8, but still, the browser did not display any Kanji characters.

At this point, what had I really accomplished? When I inserted correct UTF8 bytes for a specific Kanji into the UTF8 database, the application didn't work. Simply exporting data from the WE8ISO8859P1 database and importing into a UTF8 database resulted in corrupted (incorrect) bytes for Kanji characters. I needed to understand what the application was doing to figure out how to correct it. I believed the application should have been able to display Kanji if the correct bytes for UTF8-encoded Kanji were in the database.

How could I debug the application? The application was far too complex to attack directly. Instead, I simply avoided the problem by asking for help from one of the developers. Angela (the Java Diva—yes, she tours and has an entourage) wrote a simple Java servlet. This servlet simply retrieves the

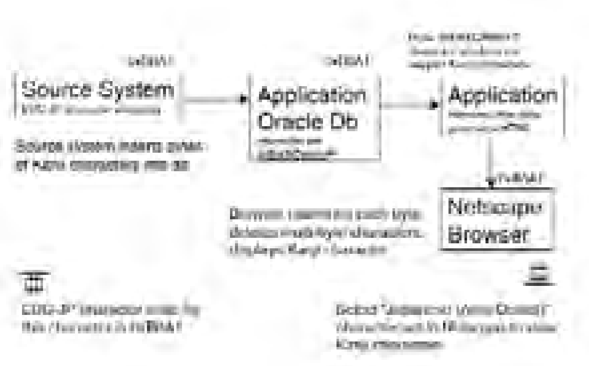

Figure 6.

bytes (character data) from the UTF8 database and then generates and sends HTML to the browser. If I could fix the servlet so that it *did* work with correct UTF8 bytes, then I would (perhaps) know how to fix the application.

I needed help modifying this servlet to work with UTF8. I appealed to the Java community and they had the answer. It turns out I needed to modify the servlet code that generates the HTML. The usual suspects were rounded up. Examples of specific modifications to the servlet code to make it work correctly with UTF8 character data are shown below.

```
res.setContentType("text/html;charset=UTF-8");
PrintWriter out = new PrintWriter(new
OutputStreamWriter(res.getOutputStream(),
"UTF-8"),true);
out.println("<META HTTP-EQUIV=" + DQ +
"Content-Type" + DQ + " CONTENT=" + DQ +
"text/html; charset=utf-8" + DQ + ">");
```

The Java code, just like the database and so many other pieces of the application, was using a default character set, which was eight-bit and used a single byte for each character. The fix was to specify UTF8 as the character set. The modified servlet code then worked correctly. The modified servlet code would retrieve the bytes that correctly encoded the Kanji character, and the correct Kanji was displayed in the user's browser using the Netscape UTF8 character set.

The next step was to fix the application. The similar portions of the application code were located and the same modifications were made. Now the modified application would work with correct UTF8 bytes and they were displayed properly. However, I then discovered that the application also retrieves web content from yet another system, and that system didn't have a character set specified so it was defaulting as well. This other system had to be reconfigured for UTF8 and everything worked.

But now the question was, had I really fixed the application? I had a modified application that worked properly with correctly encoded UTF8 data, but what about the existing data? Using the modified application, existing data (data that was exported from WE8ISO8859P1 database, imported into UTF8 database) was not displayed correctly. Further proof that the existing data was not correctly encoded UTF8 for Kanji characters.

I needed to fix the whole application, including the application data in the UTF8 database. I traced the flow of the

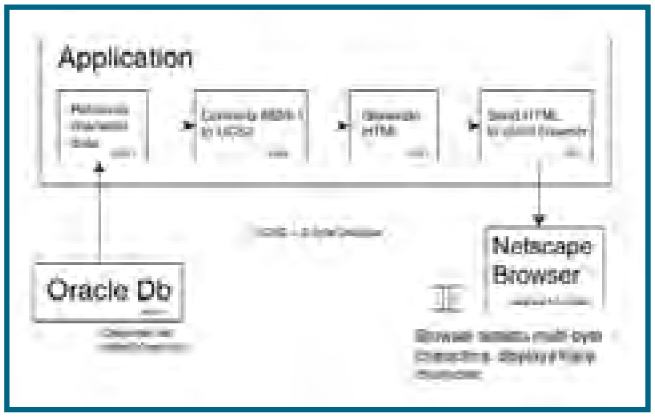

Figure 7.

existing data through the modified application, as shown in Figure 7. The four bytes that were in the UTF8 database that were the UTF8 encoding of each of the two bytes of the original Kanji were retrieved by the Java code. The Java code converts them to Unicode, which results in the two bytes that were originally inserted into the WE8ISO8859P1 database as the EUC-JP bytes for the Kanji character.

The modified application then generates HTML, but, since the application now specifies UTF8 for the character data being sent in the HTML, the two bytes in Java are converted to UTF8. The same four bytes that were in the UTF8 database for the Kanji character were sent to the browser. The browser is configured to display UTF8. The UTF8 characters displayed are the degree sign and the upside down exclamation point, very different from the correct Kanji character.

While the application itself was fixed, the existing data in the UTF8 database was not correctly encoded UTF8 character data. How would I go about fixing the existing data in the database? I had a plan. I had the application working with correct UTF8 data. The only thing left was to correct the existing data in the UTF8 database. I knew how the original Kanji data that was from the EUC-JP character set had been stored in the WE8ISO8859P1 database, and how these bytes were converted (corrupted) as they were imported into the UTF8 database. I needed some way to convert the existing bytes in the UTF8 database back to what they had been (EUC-JP) and then convert them correctly into UTF8.

It was time to review the various character set encoding schemes that we had seen during this project. The original EUC-JP data had two bytes for a single Kanji character, and there were two bytes in the WE8ISO8859P1 database. The two bytes became four bytes when imported into the UTF8 database. The correctly encoded UTF8 for the Kanji character had three bytes.

I needed to convert the existing data, but how? I could see several possibilities to fix this. Why not fix the problem in the WE8ISO8859P1 database before exporting? The problem with this is that there really wasn't much point. Whatever I did to the data in the WE8IOS8859P1 database, the export would still be an export from a single-byte database. This means import would still load one byte at a time, and each individual byte would be converted to UTF8 as it was imported into the UTF8 database. This would corrupt the bytes no matter what I did in the WE8ISO8859P1 database.

Another option would be to not export/import at all. Instead, use SQL select statements to generate a flat file for each table in the WE8ISO8859P1 database. When each of these flat files is loaded into the UTF8 database using SQL*Loader, I could use the SQL*Loader option CHARACTERSET JA16EUC, where JA16EUC is the Oracle character set that uses EUC-JP encoding. As SQL*Loader loads the bytes, they would be read in using the JA16EUC character set. The two bytes in the flat file for a Kanji character would be correctly recognized as a multi-byte character and would be correctly converted to UTF8 in the UTF8 database. The reason I didn't like this option is that I would have to repeat this for every table.

Yet another option would be to wait until after the data was in the UTF8 database. I could fix the data after the export and import process had been completed. This option is illustrated in Figure 8. This would retain the simplicity of a single export and import to get all the database objects from the WE8ISO8859P1 database into the UTF8 database. I found the Oracle SQL CONVERT ( ) function that could do this. The CONVERT ( ) function can convert character data between any two of the many Oracle character sets. The CONVERT ( ) function works on one column of one table at a time, although a single update statement could work on multiple columns at once. I did test the CONVERT ( ) function to demonstrate that it did indeed correctly convert the existing four bytes in the UTF8 data (the four bytes of a Kanji character that had been imported into the UTF8 database) into the correct three bytes.

Now I had a process to correct the data. Export from the WE8SIO8859P1 database, import into the UTF8 database, and after the import, use the CONVERT ( ) function to convert each column of each table that contained multi-byte data.

The Oracle SQL CONVERT( ) function can make one conversion at a time. I needed to make two conversions. The first conversion needed to get from the existing UTF8 character set back to WE8SIO8859P1. This had the effect of changing the data that had been imported into the UTF8 database back to the bytes as they existed in the WE8ISO8859P1 database. The second time CONVERT( ) was used to convert from WE8ISO8859P1 to UTF8. If something were to interrupt this process, it is important that CONVERT( ) not be re-used. It is easy to corrupt the

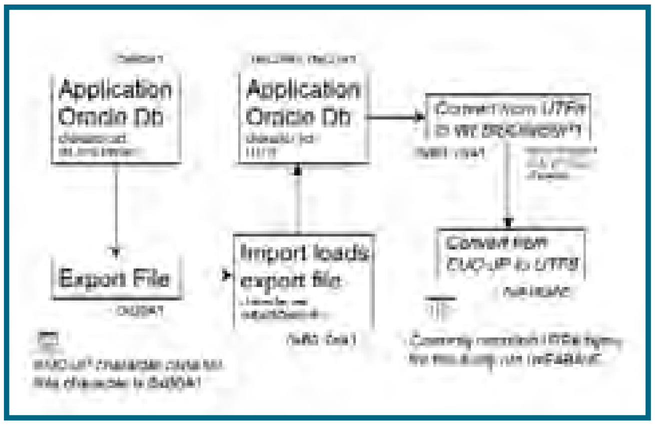

Figure 8. Fix After Import

data if CONVERT( ) is run multiple times on the same data. If needed, I would have to regenerate (export, import) the data and re-run the complete CONVERT( ) process to ensure the correct result.

With this done, I had a complete process to convert the existing WE8ISO8859P1 database to UTF8. The only additional steps were to widen the columns in the UTF8 database that would hold multi-byte data. I had seen that UTF8 could use up to three bytes to store a single Unicode character. Therefore, I widened any column that would contain UTF8 to three times the original size. It is important to realize that Oracle stores bytes, not characters. If a column was created to store 10 characters and was sized as 10 bytes before, it should be resized to 30 bytes as part of moving to UTF8.

The conversion process would then be like this: export the WE8ISO8859P1 database, import into the UTF8 database. For this import, use the ROWS=N option. This means import will create all the database objects, but won't import the table data. I would then widen the columns as needed and import again but using the IGNORE=Y option. This causes import to ignore the errors caused when import tries to create a table that already exists, and the table data would get loaded. I would then run the two-step CONVERT( ) process to correct the data.

Note that the conversion process also includes modifying the application code and any other pieces of the application that could have an affect on the character set used for processing character data.

One of several details that haven't been explained is how I knew which of the many Japanese character sets had been used to generate the Japanese character data that was in the original Sybase database. I had figured out that the original Kanji character data was from the EUC-JP character set. How did I determine this? In this crazy world how could anyone really know anything? What I did was to take a sample of the Kanji data from the original (existing) application. I identified one record in the existing Sybase database. I displayed the Kanji for this record using the original application. I then found the first Kanji character in a Japanese character dictionary (count the radicals!). This dictionary also listed the row-cell location of this Kanji in the JIS-0208 character set. Using multiple sources, I had to learn how to convert from JIS-0208 to other character sets. I generated the bytes for several of these character sets and compared them to the bytes stored in the Sybase database. This showed me that the original Kanji were from the EUC-JP character set.

Note that there is no Oracle utility that helped me with this. I had to figure this out on my own. The Oracle NLS manual provided the bytes for multiple character sets for a single Kanji, which I used as a check of my work. This was my Rosetta stone.

I needed several reference books as I worked to identify the Japanese character set that had been used. In addition to the Japanese character dictionary, I used the Unicode standard and the O'Reilly book, *CJKV Information Processing*, to learn how to do the needed character set conversions.

When it was all over, what did I learn from my relationship with Christine?

1) Oracle (and Sybase) don't store characters, they store bytes and strings of bytes.

2) Normally, Oracle does no checking of character sets, bytes inserted may or may not represent a character in the database character set.

3) Only under specific circumstances does Oracle "apply" a character se—when the NLS_LANG of the import session doesn't match the character set of the database, for example.

4) Changing character sets affects more than just the database.

5) Bytes of a character from any character set can be stored in a database created with any character set—EUC-JP data stored in WE8ISO8859P1 db, bytes aren't "correct" in the db character set.

6) Any character set conversion may corrupt the character data.

7) Simply exporting and importing to UTF8 does not solve all the problems of moving to UTF8.

8) Testing requires generating correctly encoded character data in the new character set — no utilities do this for you.

9) Every piece of an application makes some decision about the character set or has a default character set.

10) If all the existing data in a database really is in the database character set, then exporting and importing to a database with a different character set would work.

11) You need to be able to see the original data to verify the character set conversion has been done correctly — I had to know what the original Kanji looked like before the conversion — the same set of bytes can represent many different characters in different character sets.

12) Beware of interviewers in Hawaiian shirts, especially when they promise exotic locales (I was misinformed!).

There is a greater single lesson to be learned from all of this. Anytime you are changing the character set of your database, you need to realize that the existing database may contain character data that was inserted at some time in the past from a character set different than that of the existing database. In the specific case of our story here, there was Japanese character data that was originally inserted from the EUC-JP character set into an Oracle database whose character set was WE8ISO8859P1. Anytime you are converting a database to UTF8 for example, you need to decide how you will determine if all of the character data in your existing database is really from the character set that is specified for the existing database. The conversion from a character set to the UTF8 character set can be done very easily with export and import. But this does not address the general issue of whether or not your existing database contains only character data that was originally from the character set of the existing database. Using our story again as an example: the conversion to UTF8 went smoothly, but the real issue was that some of the existing character data, when it was inserted, was not coming from a source that used the same character set as that of the existing database. When the Japanese data was inserted into the original Sybase database, it was inserted from a source system that was using the EUC-JP character set. The data from that source was inserted into a Western European (ISO1, single-byte) Sybase database. Anytime you are converting database character sets, you need to take this possibility into consideration. ▲

Brian Hitchcock has worked at Sun Microsystems in Newark, California, for the past nine years. Before that he worked at Sybase in Emeryville, California. Even further into the past he spent 12 years at Lockheed in Sunnyvale, California, designing microwave antennas for spacecraft. He supports development databases for many different applications at Sun, handling installation, tuning, NLS, and character set issues as well as Unicode conversions. Other interests include Formula One racing, finishing a Tiffany Wisteria lamp, Springbok puzzles, Marklin model trains, Corel Painter 8, and watching TV (TiVo® rules!). Brian can be reached at brian.hitchcock@aol.com or brhora@aol.com.