

Creating Multi-Lingual and Multi-Locale Databases

Addison P. Phillips
Globalization Architect
webMethods, Inc.

Introduction

Providing systems integration for global companies requires more than a commitment to internationalization. While many information systems today are internationalized, increasingly companies and organizations operate on a global scale and need to integrate business practices and systems to suit this new scope. This requires systems that are capable of supporting users in more than one locale at a time.

A key component of many systems is the data store, which is often a SQL-based RDBMS. System designers often have difficulty adjusting schema designs to multi-lingual requirements, in part because there is no information on specific database idiosyncrasies and in part because there is no information on the practical concerns for how to adjust columns and tables.

This paper provides an introduction to this topic based on lessons learned in the creation of a global, web-based system at webMethods for trading partner profiles, a key step in establishing an electronic business-to-business trading relationship.

In our system, each customer manages their own trading network or “hub”. They’ll then use their hub to sponsor different B2B trading opportunities, which are called “initiatives”. Interested third parties (“suppliers”) can be directed to or sign up for a specific initiative using the on-line system.

A key element of an initiative (and the focus of this paper) is the questionnaire that the customer uses to qualify potential suppliers. Below we’ll examine how webMethods organized the database schema of a portion of this system.

Basic Problems

When designing the system, we needed to take into account the business requirements of our customers, mostly Global 2000 companies. Since the questionnaires are user-defined, they can be authored in any language. And suppliers may come from a variety of locales. But the system is centrally managed. We needed to provide:

1. The ability to store multilingual data without having separate database instances.
2. The ability for a trading network to localize a questionnaire and serve it according to the supplier’s language preferences.
3. The ability to customize the questionnaire depending on the locale of both the supplier and the hub owner (since different locations, for example, might have different requirements for the same general initiative).

Ignoring the complexity of negotiating locale and creating a user interface to allow customer to manage the above scenario, let’s look at the multi-locale database.

It turns out that there are “rules of thumb” that can be applied generally to the problem of supporting multi-lingual data. For our project, these fell into four categories:

- (1) Expand fields to support changes in character encoding.
- (2) Expand fields to support differences in the storage requirements of other languages (cultural or linguistic expansion, as opposed to encoding)
- (3) Identify if each field contains locale-neutral, locale-intrinsic, and locale-related data and normalize the tables accordingly.
- (4) Structure tables to support locale-specific ontological or logical structures to provide consistent, language-independent navigation.

Character Encoding

A relational database generally is described in terms of “data description language” (DDL). If your data requirements are somewhat complex, you probably have already encountered the problem of maintaining separate DDL for each brand of database that your product supports due to proprietary “performance enhancements” or features. It may surprise you to learn that you also must adjust the schema on a per-database-brand basis, depending on where you intend to deploy your database.

Most data types used in databases are internationalized by design and thus do not require adjustment when creating a multilingual system. Objects like datetime, integer and floating point numbers are language neutral and the database usually provides support mechanisms for formatting and manipulating these data elements according to your code’s needs.

Binary data structures (such as BLOBs) are “language neutral” in terms of how the database manages and retrieves them. It’s the responsibility of the developer to ensure they are truly internationalized. (Character large objects –CLOBs—are usually reliant on the database character encoding).

Character data (such as char and varchar2 data types), however, is stored in a particular character encoding that is usually set at database installation time. For “Latin” language encodings (those that use a Latin based alphabet such as English, French, German, etc.), each character is represented by one byte. So a varchar(30) can store thirty characters. Programmers and database designers often rely on the database’s ability to truncate at “thirty characters”, though, to skip over certain input checks, such as checking for maximum length. Or they may rely on the DDL to create length-checking algorithms in code (this is one problem we encountered at webMethods).

When we want to begin storing “multibyte” language data, such as Japanese, Chinese, or Korean, this begins to break down. For example, a varchar2(30) can store somewhere between 15 and 30 characters in the common Japanese encoding Shift-JIS. That’s because this legacy encoding’s characters are either one- or two-bytes long.

The problem with any legacy encoding is that it is tied to a specific language or writing system. If we expect our database to work globally with minimal modification and if we expect to store multi-language data at any time, we need to use a Unicode encoding.

Most of the major databases provide a Unicode encoding. An exception to this is the popular open source product MySQL, which doesn’t support Unicode encodings directly.

UTF-8

The most common encoding of Unicode used in databases is UTF-8, a multibyte encoding of Unicode. Oracle, Sybase, DB2, and Informix, for example, all provide UTF-8 implementations. UTF-8 uses one byte for 7-bit ASCII characters. All other characters are a multibyte sequence of between two and four bytes, with four byte sequences being exceedingly rare.

This means that, in order to ensure that we can store a 30 character sequence we need to have a minimum of at least 90 bytes of storage allocated.

Note that the “multibyte” nature of UTF-8 means that we could store up to 90 valid UTF-8 characters in a `varchar(90)`, if the characters happen to all be ASCII. So in-code size validation becomes important. Another reason is that we’re assuming that we will never store any of the “exotic” four-byte sequences!

One note: don’t “lie” to the database about the actual encoding used. A common mistake is to set the database up to use ISO-8859-1 as the encoding, but actually store UTF-8 sequences in the database. Since the system doesn’t “know” about the substitution, you can get odd behavior, such as partial character loss. You also lose the database’s in-built collation and comparison capabilities, since these are derived (partially) from the encoding.

UTF-16

The other common encoding for Unicode in databases is UTF-16 (or the very similar older version of this encoding, called UCS-2). In UTF-16, every character is 16-bits long (except for characters in the “supplemental planes” of Unicode: more about those in a minute).

The only major databases in full release that support UTF-16 (or UCS-2) are Microsoft SQL Server 2000, IBM DB/2 UDB (IBM CCSID 13844), and Oracle 9i. And a few “small” databases provide support for 16-bit Unicode also, such as the open source Java database Cloudscape.

Other databases are expected to add support for UTF-16 over time. For example, Sybase ASE is supposed to provide this support soon via the addition of a new data type.

The way that UTF-16 is implemented in databases differs from the way that UTF-8 is implemented. UTF-8 is “just another encoding” and applies to `char/varchar` datatypes. UTF-16 support, by contrast, is usually associated with the `nchar` and `nvarchar` datatypes. This means that the database schema will differ from the UTF-8 schema.

UTF-16 Access Problems

One aspect of this issue that confronted our project was accessing UTF-16 data from our Java programs. It turns out that most JDBC drivers cannot distinguish between a `char` and an `nchar` (for example), and therefore most existing JDBC drivers cannot retrieve data from “n” fields. The middleware solution we had selected (Merant) provided no relief, and thus our deployment plans had to be altered to exclude Microsoft SQL Server 2000, since Microsoft provides *no* UTF-8 implementation, only UTF-16 and only via the “n” types.

Future versions of each of these products are supposed to provide access to the actual schema information, and thus allow this problem to be overcome. It should be noted that similar problems exist with other UTF-16 implementations (such as Oracle 9i when used with the 8.x JDBC drivers), so you should carefully evaluate the technology you have selected to ensure that you can access your Unicode data successfully! Often this information is difficult to obtain.

Oracle 9i and UTF-16

One of the reasons that older JDBC drivers had problems with UTF-16 was that most databases were set up on what is called “client makes right” (CMR) encoding conversion. That is, the SQL statement is converted on the client side before being transmitted to the server. This means that a valid Java String object containing your SQL statement might be converted to a local legacy encoding.

In order for Oracle 9i to switch to or add the nchar equals UTF-16 style of Unicode support, Oracle needed to add support for the nchar fields. They did this by switching their newer JDBC driver to use “server makes right” (SMR) conversion: your SQL statement is transmitted to the server as a Java String (still encoded as UCS-2), and the database makes the necessary conversions for you. Their manual says the following:

“The JDBC server-side internal driver is running in the server, all conversion are done in the database server. SQL statements specified as Java strings are converted to the database character set. Java string bind or define variables are converted to the database character sets if the form of use argument is not specified. Otherwise, they are converted to the national character set.”¹

It should be noted that the newer JDBC driver is only supplied at the time of this writing for the Solaris operating system and webMethods hasn’t tried it yet, so I cannot report on the efficacy of this.

Oracle and UTF-8

As mentioned previously, database vendors have each defined their own support for Unicode. Of particular importance (and interest) is Oracle’s support.

Oracle has supported the Unicode effort from very early in the history of the “universal character set”. This support is now in its third generation (just as Unicode is now in its third “generation”). Each generation of Unicode support in Oracle is represented by a different database character encoding.

The earliest encoding is called “AL24UTFFSS”. “AL24UTFFSS” was based on a forerunner of the current UTF-8 encoding of Unicode called FSS-UTF (“file-system safe Unicode transformation format”) and supports the Unicode 1.0 standard. This encoding is now thoroughly obsolete, especially since it maintains the old mapping of Korean Hangul characters. It is only maintained for compatibility with older Oracle databases.

Later, when support for Unicode 2.0 was added, Oracle provided support in the form of a new encoding. This encoding was called “UTF8” and “UTF8” is the Unicode encoding used by Oracle 7, Oracle 8, and Oracle 8i. For a variety of reasons, the “UTF8” provided by Oracle is not strictly the same thing as the UTF-8 defined in the Unicode standard. Basically, a decision was made to limit “Oracle UTF-8” to the characters in the Basic Multilingual Plane (BMP) of Unicode. Since until recently these were the only defined characters in Unicode, nobody experienced any problem with this.

Unicode 3.1 finally adds some characters beyond the BMP. As you might expect, Oracle has provided a new encoding to support these. It’s called “AL32UTF8”.

The difference between “UTF8” and “AL32UTF8” is that “AL32UTF8” stores characters beyond U+FFFF as four bytes (exactly as Unicode defines UTF-8). Oracle’s “UTF8” stores these characters as a sequence of two UTF-16 surrogate characters encoded using UTF-8 (or six bytes per character).

This means that Oracle’s “UTF8” violates the “shortest form” requirement in the Unicode standard’s UTF-8 Corrigendum and it makes the “UTF8” encoding of Oracle into something other than “UTF-8”. Obviously this is going to confuse a lot of people.

In addition to being subtly different in the way that it handles some characters, you should note that the two encodings provide different sort sequences (since most sort sequences for non-BMP data will be binary sort on code points) and, of course, the six bytes per character take up a lot of valuable space in a varchar2!

¹ http://download-west.oracle.com/otndoc/oracle9i/901_doc/server.901/a90236/ch6.htm#1005692

It also means that the raw byte stream from an Oracle “UTF8” database needs to be re-encoded as “real” UTF-8 before, say, posting it to an external process that expects “corrected” sequences. For the most part, applications (such as our Java program) that retrieve data from Oracle will probably interpret the bytes into their own Unicode format anyway, so the most noticeable effect will be differences in sort sequence (and, of course, available storage). But it is an implementation issue to be aware of as Unicode 3.1 support percolates into operating systems and environments. If nothing else, you may want to ensure that your future implementations use “AL32UTF8” if possible.

Finishing Up with Encodings

In sum, you need to change your database schema to match the actual storage requirement of your data by multiplying the number of characters you are *required* to store by the maximum number of bytes in the encoding. And you should prefer a Unicode encoding for your database over local legacy encodings if your intention is to support multiple languages in a single database instance. Finally, I should note that for UTF-8, the number of bytes multiplier can probably stay at three for some period of time, but consideration should be made of the newly added characters in Unicode 3.1.

Cultural Data Expansion

Once an encoding has been selected and the schema has taken the expansion into account, we still have to deal with handling the requirements of different locales. Selecting an encoding provides us with the opportunity to store characters from languages other than English, but we still don’t necessarily have enough room to store data from countries other than the USA.

In addition to the expansion of char and varchar fields due to the character encoding, we also need to review the various database fields to ensure that they can store the likely length of overseas data.

One of the tables in our project stores the shipping and mailing addresses of “trading partners” and their specific contacts. In the original table, designed with the U.S.A. in mind, was a field called “State”. The database defined “State” as a char(2) [fixed length, 2 bytes]. The field was related to a “states” table that provided referential integrity for states of the USA, American territories, and Canadian provinces. Addresses outside the USA had to store equivalent data in different field. Obviously it is poor design to store the same abstract data values in different fields and the code behind this field was very USA-centric. No validation can be provided, for example, for addresses outside the USA.

Modifying the table to allow for addresses from multiple locales means changing the format of the field to provide enough storage for the largest provincial name or code that we intend to allow.

Similarly identification numbers, addresses, textual descriptions, and the like tend to expand in the same way that localized user interface messages expand. Generally speaking, free text expands by about 30% from English with “translation”. Some expansion requires much more than this, though, because the nature of the data changes.

For example, in the original database, the system provide for contacts to have a first name, last name, and middle initial. Obviously it is a simple thing to omit the “middle initial” field for countries where it isn’t necessary or is culturally inappropriate. However in some countries it is customary to store much larger names or combinations of names in this field than just an initial. The original field was a varchar2(10)—more than enough to store an “initial” even if the initial were a Han character in UTF-8, but we expanded it to be a varchar2(90).

Some fields, of course, have a specific length because the underlying data is presumed to be a certain size (like the state example earlier). These assumptions must be identified in both code and the database and removed or modified to fit the likely range of variation in the field worldwide. As we’ll

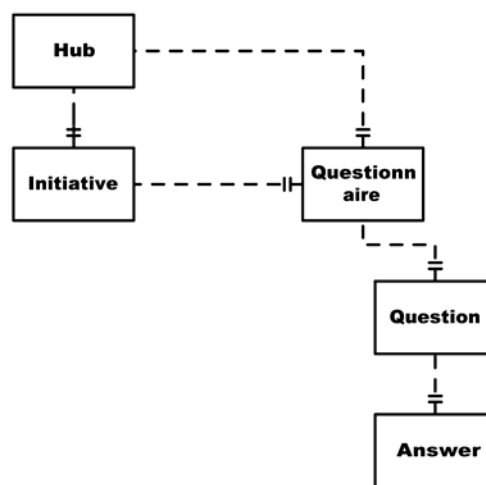
see later, locale-sensitive code is better at adapting to (and validating) some of these features than our static database tables are.

There is also an obvious tension between the desire to make the fields big enough and the desire to make the database efficient. Making all of the text fields into varchar2(255) is wasteful, so care must be taken to ensure that this is done sensitively.

Normalize to Allow Localization

So far we've provided enough storage for any specific data field, with fields expanded either because of encoding or for cultural (locale based) reasons. Isn't that enough? Not quite.

For one thing, we found that our data fell into three distinct groups that required different handling. These are "locale-neutral", "locale-intrinsic", and "locale-related". In order to understand these different types, let's look at a simplified model of the questionnaire schema:



In the diagram above, each hub has a "joining" questionnaire and a series of initiatives. Each initiative has a questionnaire. Each question consists of a set of questions, each of which may have a set of accepted answers.

Locale Neutral Data

Locale neutral data seems like the easiest to identify. It's data that is invariant and whose display is unaffected by locale, language, country or other regional variation. The "hub ID number", for example, is locale neutral. That one ID identifies a specific hub record—and only one hub record—regardless of the locale of the user, and the ID isn't formatted, validated, or displayed differently depending on, for example, the supplier's locale.

Locale Intrinsic Data

By contrast, locale intrinsic data has some locale preference "embedded" into it. The locale of the data affects how it is displayed, processed, or validated. For example, if we were to store the hub's mailing address in the Hub table, the address fields have an intrinsic "locale" that affects their formatting, display, and entry validation: the country in which the address is located implies the locale of the data.

Like locale neutral data, though, locale intrinsic data doesn't require a special storage mechanism or a big change in the database schema. The locale information may need to be stored (so that it doesn't have to be inferred from the data), but generally, what's important is recognizing that the locale is implied by the data.

Locale Related Data

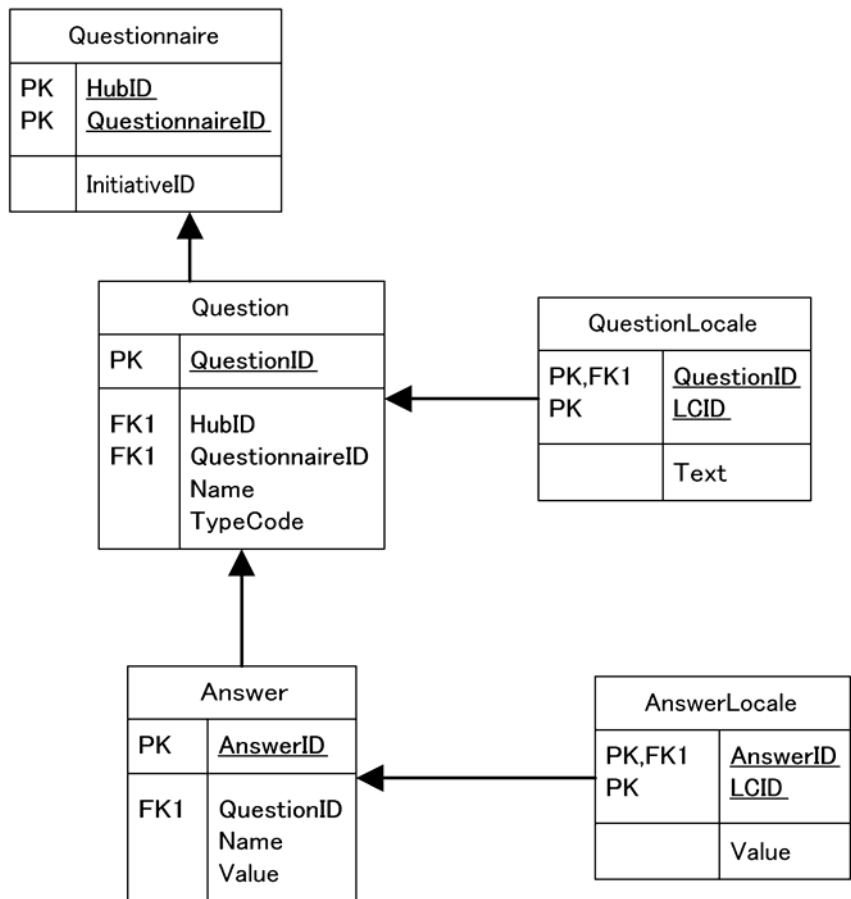
Locale-related data changes when the user's locale changes. Usually this is something defined internally that we want to provide multiple versions of, depending on the language preference of the end user.

This is one thing that made the questionnaire project interesting. We knew that the hub owner would create initiatives and then want to offer that same initiative in different circumstances—for example, purchasing supplies for different facilities in different locations. There are two ways we could accomplish the roll-out of an initiative across language boundaries:

- 1) Make the hub owner recreate the initiative in the new language.
- 2) Allow the hub owner to localize the existing initiative by providing “slots” in the database for the locale-related data.

The problem with the first solution is that it makes it difficult for the hub owner to see if a particular initiative is being successful or to look at the results of answers to a specific question. So we need to provide for a way to refer to localized content that is invariant across locales (in this case the text of the questions and the acceptable values for the answers).

One way to do this is illustrated in Figure 3 below. We created locale-based tables to hold “slots” for the translation.



Most of the data in the Question and Answer tables consists of IDs, binary identifiers that have no intrinsic locale or textual value, even though they happen to be “char” fields. We don't need to worry about these, because the IDs will be generated in code and are not intended to be human readable.

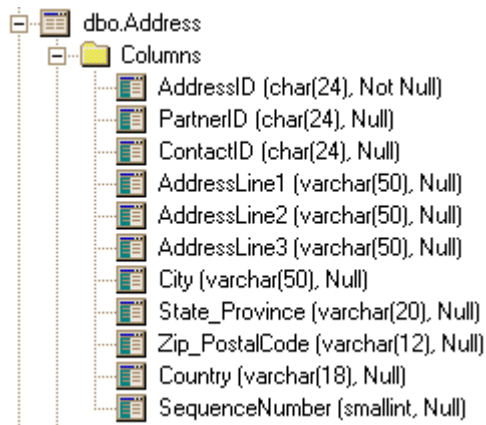
Neither do we need to worry about any datetime or number data types: we'll format this binary data using a formatter at runtime.

The textual data that ends up being displayed to the end user is stored in the locale-based tables (in the diagram you see IDs as the Foreign Keys, in practice we used Oracle ROWIDs to speed access). In this case we separated the text of the Question and answer values.

Of course, more than just text can vary by locale. For example, the range of numbers in an Answer might need to vary by locale.

Locale Dependent Data

Let's look at a subtler example. As mentioned earlier, an Address table has an intrinsic "data locale":



We've already discussed the fact that different data fields here need to be expanded to accommodate encoding expansion (due to the use of non-ASCII characters) and data expansion (due to longer requirements of some overseas addresses). What isn't immediately apparent is that the format of the screen used to collect this data is related to which country you are collecting data for.

In the first table design (above), "country" is a free text field that allows up to 18 bytes of storage (not enough for "United States of America"). It was the last data entry field collected in our prototype, too, so we couldn't use it to affect the way that the address was collected and its data validated.

Instead we might set up the table to use the ISO 3166 country code. It's important to note that a country is not a locale. The ISO 3166 codes are used as part of a locale definition, but the data relationship here isn't to a locale, but to the specific country.

Here's the prototype entry screen:

Address 1

Address: 1111
1 2 3

City: dkdkdk

State/Province: dkdkdk

Postal code: dkdkdk

Country: USA

Here's how it might look with different data in an internationalized version:

Address 1 (USA)

Country: USA

Address: 1111
1 2 3

City: dkdkdk

State/Province: [dropdown]

ZIP code: [input]

Address 1 (Japan)

Country: 日本

Postal code: 1111

State/Province: 1 2 3

City: 1111

Address: 1 2 3
1111
1 2 3

In this example, the code can react to the selection of a Country and display the proper screen format and validation behavior

Obviously we didn't support every country on Earth—there are 192 recognized by the International Postal Union²—but we can provide logical behavior for the 30 or so countries that most interest us, support for expansion without adding code, and good fallback control for those that we don't support with a “generic” address form.

What's interesting about an address is that it is a collection of fields that are locale (or, in this case, country) related. Although this database appears to have a separate table for addresses, the real database actually has these fields embedded in appropriate partner and contact tables.

What's really interesting about this aspect of the product's mailing address component is that there is the concept of user locale in the same tables with the addresses, since each contact and system user needs to select which language version of the questionnaire (for example) they wish to have displayed and this information isn't implied by the country of their mailing address.

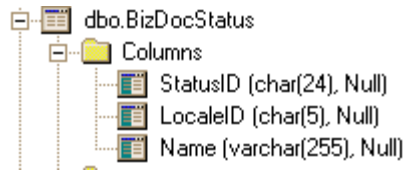
Another Example

Let's look at another example of the same concept. In the table BizDoc we store a number of Status values that indicate where the supplier is in the process of completing the questionnaire and how they had been scored. These were predefined in our source code in the prototype system, but maintained as free text fields (varchar2(255)) in the database.

² <http://www.upu.int>

The problem with this, of course, is that the status was visible to the end users because the text fields were meaningful English phrases. We want to make the status locale-neutral, but provide for localization.

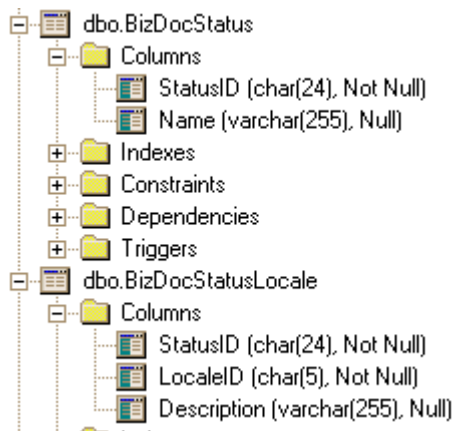
We can fix this by replacing each free text field with another ID-type field and creating a table to contain the statuses. Such a table might look like this:



This looks good: we can create different status IDs that have a different description based on the “locale ID”.

The problem with this is that each locale has a separate set of status IDs. If we want to query the system to find all the BizDocs with a status of “OPEN”, we have to perform several queries or an expensive join.

So what we actually need to have is two tables, BizDocStatus and BizDocStatusLocale:



Now we can create a locale independent object (a “status”) with an “English-like” descriptive value (the “name” field), which can be localized in a variety of ways. We can query the system for “OPEN” documents by having the user click on the localized “Description” but performing the query based on the StatusID.

This kind of two-table “ontology” can be used in a variety of ways in the system to ensure that locale-dependent descriptions and other *internal* data can be localized (in much the same way that ResourceBundles are used to extract the hardcoded strings from software).

We adopted the field names “Name” and “Description” very deliberately, as well. A “Name” we defined as the text field stored in the locale-neutral file. You can think of it as the “default locale text” for the object defined in the table. Generally, the “Name” is in the language of the data’s original author.

The “Description” field we defined to be the localized representation of the data. It’s always the one associated with the LocaleID field.

While the tables above are quite simple, it turns out that many tables contain a large number of locale sensitive fields. In creating a complex system, you’ll often find that you want to use the user’s locale to

limit the data displayed or change the way that it is organized. The actual “ontology” of the data may be different. By adopting this specific terminology, we were able to make clear whether the table contained “the actual data” or just a (locale variable) representation of that underlying data.

Something else you might notice about the tables above. The “LocaleID” field suggests that there is a table called “Locale” somewhere. In fact, we created a table that tells us which locales are supported and installed on this system. This allows the system to present choices about what formatting and localization is available on a particular system instance. By keeping this information in the database instead of in resource files, we made it easier to distribute and install new “locales” to customers.

Locale Intrinsic Revisited

So far we’ve dealt with specific fields within a table. We’ve identified expansion due to encoding and cultural requirements, and we’ve provided for the localization of locale-variant data.

One other feature of establishing and maintaining a trading relationship is the need to create a custom methodology—the steps the supplier needs to go through to become accepted in a B2B trading relationship. Unlike the questionnaire, which customers often wish to recycle as part of a successful initiative, methodologies are quite time-consuming to set up and are often driven by regulatory, legal, and human interface issues.

As a result, the “Methodology” table has a locale defined in the very top table in the hierarchy. The myriad of lower level tables in the methodology structure now have an intrinsic locale implied by the top level methodology.

This means that there are no localization “slots” in the tables. It also means that the methodology is tied to a specific display language.

Summary

Designing a multi-locale database system, while challenging, is relatively easy to accomplish, provided the developers select an appropriate encoding, expand storage to accommodate the encoding and cultural changes to the data, and structure the data to provide for multiple language representation of locale-sensitive data and the tagging of locale-intrinsic data so that it can be displayed appropriately.