Migration to Unicode Datatypes for Multilingual Databases and Applications

An Oracle White Paper
January 2005

Migration to Unicode Datatypes for Multilingual Databases and Applications

Migration to Unicode Datatypes for Multilingual Databases and Applications

Introduction

Unicode datatypes were introduced in Oracle9*i*. Unicode datatypes are supported through the SQL NCHAR datatypes: NCHAR, NVARCHAR2, and NCLOB. (In this paper, "Unicode datatypes" refers to SQL NCHAR types.) SQL NCHAR datatypes have existed since Oracle8. However, in Oracle*9i* forward, they have been redefined and their length semantics have been changed to meet customer globalization requirements. Data stored in columns of SQL NCHAR datatypes are exclusively stored in a Unicode encoding regardless of the database character set. These Unicode columns allow users to store Unicode in a database, which may not use Unicode as the database character set. Therefore, developers can build Unicode applications without dependence on the database character set. The Unicode datatypes also make it easier for customers to incrementally migrate existing applications and databases to support Unicode.

This paper consists of three parts. First, it briefly introduces the Unicode datatype features. It then discusses the steps to migrate existing databases to make use of the Unicode datatypes and what the considerations are about functionality and performance. The third part highlights the steps to migrate applications to support Unicode datatypes and work with the database schema changes. Finally, it concludes with a brief summary.

Unicode Datatype Features

The concept of the Unicode datatype, introduced in Oracle*9i* allows customers to support Unicode columns in a non-Unicode database. This is a very powerful feature and is further enhanced by the inter-operability between SQL NCHAR types and other datatypes. Users can store, process and retrieve SQL NCHAR data the same as SQL CHAR data.

There are several major aspects for the new Unicode datatype features that will be discussed in this section:

- Character set encoding
- Character length semantics
- Interoperability
- Data loss handling
- Unicode string processing

Unicode Character Set Encoding

Oracle9i and Oracle Database 10g support two Unicode encodings for the Unicode datatypes. The Oracle characters set names are AL16UTF16 and UTF8. When the database is first created AL16UTF16 or UTF8 can be specified as the "national character set" parameter. When the national character set is not specified, the default is AL16UTF16. For details on Unicode encoding support, please refer to the Globalization Support Guide.

Supplementary Character is a Unicode encoded character having a Unicode code point between U+10000 and U+10FFFF. In UTF-16 encoding, it is encoded with a surrogate pair that consists of a sequence of two Unicode values, where the first value is a high-surrogate in the range U+D800 through U+DBFF and the second is a low-surrogate in the range U+DC00 through U+DFFF.

Here is an example of creating a database with AL16UTF16 as the NCHAR character set:

```
CREATE DATABASE mydb
MAXINSTANCES 1
MAXLOGHISTORY 1
MAXLOGFILES 5
MAXLOGMEMBERS 5
MAXDATAFILES 100
DATAFILE '/vobs/oracle/oradata/mynewdb/system01.dbf' SIZE 325M REUSE
CHARACTER SET al32utf8
NATIONAL CHARACTER SET AL16UTF16
LOGFILE GROUP 1 ('/vobs/oracle/oradata/mynewdb/redo01.log') SIZE 100M,
GROUP 2 ('/vobs/oracle/oradata/mynewdb/redo02.log') SIZE 100M;
```

Character Length Semantics

Character length semantics was also introduced in Oracle*9i*. It measures characters based on number of Unicode code units instead of number of bytes. - When it is based on the number of bytes, it is called byte length semantics. A Unicode code unit is a 16-bit unit that is used to represent a UCS2 code point or its equivalent in other character sets. It is the minimal bit combination that can represent a unit of encoded text. It is equivalent to the length semantics for Java strings. The advantages of character length semantics over byte semantics include lengths close to visual length and portability across different character sets. While the default length semantics for a database is BYTE, character length semantics is the only allowable length semantics for SQL NCHAR types. It is not allowed to specify any 'CHAR' or 'BYTE' quantifiers for SQL NCHAR column definitions.

Example:

```
CREATE TABLE emp (
    empno NUMBER(4),
    ename NVARCHAR2(10),
    job NVARCHAR2(9),
    mgr NUMBER(4),
    hiredate DATE,
    SAL NUMBER(7,2),
    deptno NUMBER(2));
```

The *ename* column can hold up to 10 Unicode code units. When NCHAR character set is AL16UTF16, its maximum byte length is 20 bytes. Otherwise, if the NCHAR character set is UTF8, the *ename* column would again hold up to 10 Unicode code units, but its maximum byte length is 30 bytes.

Interoperability

When operations between SQL NCHAR types and other datatypes are necessary, users can either apply explicit conversion functions or rely on the system to do implicit conversions.

- **Explicit Conversion Functions**

  The explicit conversions are more visible and controllable. Users can control the conversion direction. Oracle provides a wide range of conversion functions to meet customer requirements. Some examples of the explicit conversion functions include TO_NCHAR and TO_DATE. (Refer to *the SQL Reference* for a complete list with detailed explanations).

- **Implicit Conversion**

  Oracle supports implicit conversions between SQL NCHAR and SQL CHAR datatypes, as well as between SQL NCHAR and other datatypes such as DATE, NUMBER, ROWID, RAW and TIMESTAMP. Implicit conversion occurs whenever any operation happens between different datatypes or the type of argument is different from the formal definition. Such operations include SQL INSERT, UPDATE, and SELECT statements, assignment in PL/SQL, comparison, concatenation in SQL or PL/SQL statements, and SQL functions The implicit conversion makes migration to SQL NCHAR much easier. The following is an example in which implicit conversion happens:

  INSERT INTO emp VALUES (100, ***'Scott'****,* ***'Engineer'****, 10, '06-05-2001', 50000, 5);*

  Both the 2nd and 3rd columns in the table *emp, ename* and *job* are defined as NVARCHAR2. The literals *'Scott' and 'Engineer'* are converted to NCHAR character set encoding (i.e., AL16UTF16) before the data is inserted into the table columns.

  NOTE: Implicit conversion between CLOB and NCLOB is supported starting in Oracle Database 10g.

Exception Handling for Data Loss

If the database and NCHAR character sets are different, implicit or explicit conversion between SQL CHAR and SQL NCHAR types involves character set conversion. The only configuration that does not involve character set conversion is when both the database and NCHAR character sets are UTF8. This is not a common configuration and all the other cases will involve character set conversion. In some cases when there is no corresponding mapping character in the destination character set, a default replacement character will be used in the destination buffer and data loss occurs. Users can choose whether to be alerted when this data loss happens by setting the NLS parameter NLS_NCHAR_CONV_EXCP. The ORA-12713 error will be returned in case of data loss if this parameter is set to TRUE. Its default value is FALSE; data loss will not be reported by default. This session parameter can be specified for the whole RDBMS instance in the init.ora parameter file or in the current user session by using the ALTER SESSION statement. It can be dynamically changed during a session.

SQL Unicode String Processing

Besides the SQL explicit conversion functions' support for NCHAR, Oracle has enhanced all the SQL string functions to process NCHAR data. SQL NCHAR data can be passed to any SQL functions the same way as SQL CHAR data. SQL functions with multiple string parameters can take mixed SQL CHAR and SQL NCHAR parameters and convert them properly before processing.

Example:
SELECT LENGTH(ename) from emp;
SELECT INSTR(ename, 'SCOTT', 1) from emp;

Unicode Data Input

The UNISTR and NCHR functions allow the user to input Unicode strings in an environment or database that does not support Unicode. With the UNISTR function, the Unicode data can be represented by '\' followed by the UTF-16 code unit value of the character in hexadecimal format. NCHR returns the Unicode character having the binary equivalent to a number in the national character set.

Example:
INSERT INTO emp (empno, ename) VALUES (10, UNISTR('abc\1E05'));

All the metadata in Oracle are represented and processed in database character set encoding. For some cases in which a string literal needs to be specified in the metadata or query statement, the literal data may belong to either SQL CHAR or SQL NCHAR datatypes. The UNISTR function can be used to specify a string literal for SQL NCHAR datatypes that cannot be represented in the database character set. The following is an example.

CREATE TABLE Dept_tab (
Deptno NUMBER(3),
Dname NVARCHAR2(15),
Loc NVARCHAR2(15),
CONSTRAINT Loc_check1 CHECK (loc IN (UNISTR('\7533'),N'NEW YORK')));

The ASCIISTR function is the opposite of UNISTR. It takes as its argument a string in any character set and returns an ASCII version of the string. Non-ASCII characters are converted to the form \xxxx, where xxxx represents a UTF-16 code unit.

Unicode Data Input in Oracle Database 10g

One main purpose of the national character set is to allow you to add Unicode datatype columns incrementally to a non-Unicode database instead of migrating the entire database to Unicode. In prior releases to perform SQL DML using `NCHAR` literals you had to use either `UNISTR`, which added complexity or the data had to be compatible with the database character set. Hence you needed to have the entire database as Unicode in order for all literal data to be converted properly to `NCHAR` columns. The `NCHAR` literal string support in Oracle Database 10*g* R2 allows you to apply DML to specific data strings without having to use either `UNISTR` or converting to the database character set.

The Oracle Database 10*g* R2 provides an environment variable `ORA_NCHAR_LITERAL_REPLACE`, which, when set to `TRUE`, allows DML statements to use N-Quote (N'') without any data loss. This means that multilingual data can be added without restrictions such as having to provide hex Unicode values. The support for this feature is available in SQL, PL/SQL, OCI, and JDBC.
U-Quote(U'') is a new string literal for Unicode also available in Oracle Database 10*g* R2. This string literal takes any character in hexadecimal format of its Unicode encoding such as `U'\1234'` just like the string literal passed to `UNISTR()` function. This string literal is converted into database `NCHAR` character set as `NCHAR` type. Because every character has a corresponding Unicode encoding, all data is safely shipped to the server side without any loss. The size limitation of U-Quote is 20K bytes and the error of size limitation is only given when it

is converted into `NCHAR` if it is over the `NCHAR` size limitation. . U'' is independent of the environment variable `ORA_NCHAR_LITERAL_REPLACE`.

Limitations

SQL NCHAR types can be used almost the same as SQL CHAR types with the following exception:

- Oracle Text does not support SQL NCHAR types.

Database Migration

Even though Oracle provides a rich set of features to make the migration process simpler and less work, there are a number of issues that need to be taken care of during migration, ranging from choosing the Unicode character set to schema migration. In this chapter, we will talk about the detailed steps to migrate a database schema as well as some of the database features that can be used to assist the migration process in each step.

SQL NCHAR types in Oracle*9i* and Oracle Database 10g are significantly different from the SQL NCHAR types in Oracle*8i*. If there are SQL NCHAR columns in Oracle*8i*, they must be migrated to Oracle*9i* NCHAR semantics when the database is upgraded to Oracle*9i* by running the migration scripts that are included. Without migrating the old SQL NCHAR columns, they cannot be accessed in an Oracle*9i* or Oracle Database 10g database server. This migration step is mandatory to bring the database to a consistent state. The discussion in this article does not focus on such migration because no major issues are expected. The more common migration scenario is to migrate SQL CHAR types to SQL NCHAR types which will be discussed in more detail below.

Step 1. Choosing the Unicode Character Set

Both UTF-16 and UTF-8 can be chosen as the encoding format of the Unicode data types. This enables the database server to support UTF-16 encoding natively. For those applications, which use UTF-16 as the internal encoding, they can work with the database server more seamlessly and more efficiently. The choice between UTF-16 and UTF-8 encoding allows data to be stored on disk more efficiently as Asian and western customers can choose the encoding that is most efficient and compact for their data.
How to pick the character set for Unicode depends on the data that the application is dealing with and the nature of the application itself. Consider these factors:

- **Space efficiency:** UTF8 is more compact for data from English and European languages and less compact for data from Asian languages. AL16UTF16 is more compact on data from Asian languages and less compact for data from English and European languages. More compact data storage will result in less memory usage and disk space savings and less disk I/O, thus improving processing speed.

- **String processing speed:** AL16UTF16 string processing is generally faster.

- **Supplementary Characters Support:** AL16UTF16 supports supplementary characters while UTF8 does not support supplementary characters.

- **JAVA or Windows Application:** UTF-16 is the encoding form for Java strings and the Windows environment. There is no conversion when NCHAR data is exchanged between the RDBMS and Java strings.

A developer should consider the above factors when they decide which character set encoding to choose for the Unicode datatypes. AL16UTF16 encoding is the default encoding. If the majority of the data is not ASCII or the space requirement advantages of using UTF8 are not very strong, AL16UTF16 is recommended over UTF8. Since UTF8 can be used as the database character set, the requirement to use it in SQL NCHAR datatypes is not as strong as AL16UTF16.

Step 2. Identify Columns to be Migrated

The Unicode datatypes allow users to do either partial or full migration to support Unicode depending on customer needs. If only a limited number of columns are needed to support Unicode or you would like to do a gradual migration to Unicode, then the Unicode datatypes should be used. To do a full database migration requires all the character columns outside of the SYS and system schema to be changed to SQL NCHAR datatypes. In this case, migrating the database character set to support UTF-8 may be a better option than using Unicode datatypes.

There are various reasons that a user may want to partially migrate the database schema to Unicode datatypes. One scenario is that there is no immediate need to do the full migration. Such need may arise in the future. To migrate the whole application schema certainly involves more work and possibly longer system downtime. In this case, it may be preferred to do incremental migrations, and the Unicode datatype features can accomplish this. Often certain schemas correspond to certain applications. Some of these applications are required to support multilingual data. Such requirements may not be true for some other applications yet (even though there might be such requirement in the future). Therefore there is no immediate need to migrate the corresponding schema. User can choose to migrate the required schema's first and migrate the remaining schema's when there is such a requirement.

Step 3. Considerations Before Migration

The following discusses some considerations before the actual migration is performed.

Constraints that are defined on the migrating columns may be violated if the new definition fails to conform to the constraint's rule. For example, suppose a primary key constraint is defined on the modified column. After the migration, it could happen that multiple different key values are converted to the same key values. This is possible because the mapping between a native character set and Unicode is not necessarily a one -to-one mapping and may be a multiple-to-one mapping. The byte length of the modified column is also changed during migration due to data expansion or shrinking. If the constraint involves the byte length of the column, it maybe violated as well. The solution is to drop the constraint before migration. After the migration, whether the constraint should be activated again depends on the specific user's requirement and should be analyzed case by case.

For a column with a reference constraint to another table, both the original and the referenced tables need to be migrated at the same time. A SQL NCHAR column cannot reference a SQL CHAR column or vice versa because the corresponding columns in the dependent and referenced tables must use the same datatype.

Maximum Column Sizes: The maximum allowable sizes for CHAR/NCHAR or VARCHAR2/NVARCHAR2 columns are 2000 and 4000 bytes. A single-byte character in a database character set is mapped to a 2-byte character in AL16UTF16 or even a 3-byte character in the UTF8 character set. Because of the possible data expansion, the maximum size limit may be broken after all the character data is converted to Unicode. User should be aware of this possible violation of size limit. Depending on the actual requirements, user may need to use NCLOB as the datatype instead.

Triggers that are defined on the columns that are modified to be Unicode datatypes may be activated if specific triggering conditions occur. The conditions may include UPDATE, ALTER, CREATE, and so on, depending on how the trigger is defined. Developers should be aware of these side effects and may need to disable the trigger to avoid unexpected effects.

Partition: It is more complex to migrate partitioned columns to Unicode datatypes because a column's character encoding (and therefore the binary storage format) and length semantics will be likely changed after migration. The partition based on the old binary storage format will no longer be valid. The partitioned tables need to be repartitioned based on the new character set encoding and length semantics. We recommend using the export and import utilities to migrate the partitioned tables following the procedure below. Please consult the *Database Utilities* manual for more detailed information about the export and import utilities.

1. Export the tables that are to be migrated.
2. Alter the column definitions to be Unicode datatypes.
3. Import the table data to the modified table.

The Character Set Scanner (CSSCAN) is specifically designed with database character set migration in mind and can create a report which helps to estimate the time and effort required to migrate. In most cases, any migration method should always begin with analysis of the reports that CSSCAN produces. Specifically, one can set the source database character set to whatever is native and set the target database character set to UTF8 or AL16UTF16 to simulate the migration from CHAR to NCHAR. For more information on the Character Set Scanner, please refer to the *Globalization Support Guide*.

Step 4. Schema migration

Before the application is migrated to Unicode datatypes, a user needs to consider schema migration. Several approaches can be taken for schema migration. The following are the different approaches for schema migration and the comparisons among them.

Apply 'Alter Table' SQL Commands

The ALTER TABLE statement can be used to change an existing SQL CHAR column to a SQL NCHAR column or to add SQL NCHAR columns to existing tables. The user can also define totally new tables with SQL NCHAR columns if needed. We will discuss the scenarios and procedures suitable for each case.

- Does the new Unicode data to be added match the purpose of the existing SQL CHAR table column? Do not try to put address information into an existing name field, for example. Otherwise, if the new Unicode datatype is to support multilingual data that cannot be covered by the existing database character set, consider migrating the existing SQL CHAR columns to SQL NCHAR.

Example: A global company A originally focused business in West European and America. They have a database with the WE8ISO8859P1 character set. The NCHAR character set is AL16UTF16. An employee table is defined as follows:

```
emp (
    empno NUMBER(4),
    ename VARCHAR2(10),
    job VARCHAR2(9),
    mgr NUMBER(4),
    hiredate DATE,
    SAL NUMBER(7,2),
    deptno NUMBER(2));
```

Company A expands to Asia and the new Asian employee information needs to be stored in the database. The current database character set, WE8ISO8859P1, does not include Asian characters. The Asian employee information cannot be stored in the existing *emp.ename* column. There is no need to add additional columns to the *emp* table since the employee name data is not a new category. The best

solution is to alter the *emp.ename* column from VARCHAR2(10) to NVARCHAR2(10). This not only solves the character set encoding issue but also changes the length semantics of the column from 10 bytes to 10 characters. Thus it avoids a potential truncation issue due to multibyte characters needing more than 1 byte per character.
The following SQL command can be used to change table column definitions from SQL CHAR types to SQL NCHAR types:

ALTER TABLE emp MODIFY (ename NVARCHAR2(10));

This ALTER TABLE command not only changes the column definition from SQL CHAR type to SQL NCHAR type, but also converts all the data in the column from the database character set to the NCHAR character set. Note that CLOB columns cannot be modified to be NCLOB using the ALTER TABLE command. The *online table redefinition feature* can be used to change a column from CLOB to NCLOB. Please refer to the section "Online Table Redefinition" for a detailed description.

If there are any indexes built on the migrating column, dropping the indexes can speed up the ALTER TABLE command because indexes are updated when each row is updated.

NOTE: The maximum column lengths for NCHAR and NVARCHAR2 are 2000 and 4000 bytes. When the NCHAR character set is AL16UTF16, the maximum sizes for NCHAR and NVARCHAR2 columns are 1000 and 2000 characters, which are 2000 and 4000 bytes. If this size limit is violated during migration, consider changing the column to NCLOB instead.

- If the new Unicode data has a different purpose than the original column was designed for, it is not suitable for the existing SQL CHAR columns to hold the new Unicode data. It is not necessary to create a whole new table just for the Unicode data. Consider whether it can be fit into any existing table. For example, in some Asian countries, one's hometown or native place is important information, but in this scenario, there is no existing column that can reflect this information. Then it is appropriate to add an additional column of Unicode datatype to the existing *emp* table. The following SQL command can be used to add a Unicode column to the *emp* table:

ALTER TABLE emp ADD (org NVARCHAR2(10));

The new table definition is following.

emp (
empno NUMBER(4),
ename VARCHAR2(10),
job VARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
SAL NUMBER(7,2),
deptno NUMBER(2),
org NVARCHAR2(10));

- The Unicode columns are totally new and their purposes are much different from the existing SQL CHAR columns. There is also some other information associated with the Unicode data. This requires a new table with Unicode columns. For example, company A also needs to keep track of the employee's permanent address. The following command can be used to define a table with Unicode columns:

CREATE TABLE addr (
empno NUMBER(4),
street NVARCHAR2(50),
city NVARCHAR2(10),

```
state NVARCHAR2(10),
country NVARCHAR2(10),
 zip NUMBER(6));
```

Online Table Redefinition

For a large table with a huge number of rows that need to be migrated to Unicode types, it takes significant time to convert all the data in the column to Unicode. During this process, all the column data will be unavailable for either read or update operations. The Oracle online table redefinition feature can be used to significantly reduce the down time required to do the migration. Using this feature, the table is accessible to DML during much of the migration process. It is locked in the exclusive mode only during a very small window that is independent of the size of the table and the complexity of the redefinition. The following are the steps to migrate to Unicode types using the online table redefinition feature.

1. Verify that the table can be redefined online by invoking the `DBMS_REDEFINITION.CAN_REDEF_TABLE()` procedure. If the table is not a candidate for online redefinition, this procedure raises an error indicating why the table cannot be redefined online. For example, *scott.emp* table is being migrated.

```
EXECUTE DBMS_REDEFINITION.CAN_REDEF_TABLE('scott', 'emp');
```

2. Create an empty interim table (in the same schema as the table to be redefined) with SQL NCHAR types as the desired attributes.

```
CREATE TABLE int_emp (
empno NUMBER(4),
ename NVARCHAR2(10),
job NVARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
SAL NUMBER(7,2),
deptno NUMBER(2),
org NVARCHAR2(10));
```

3. Start the redefinition process by calling `DBMS_REDEFINITION.START_REDEF_TABLE()`.

```
EXECUTE DBMS_REDEFINITION.START_REDEF_TABLE('SCOTT',
'emp',
'int_emp',
'empno empno,
to_nchar(ename) ename,
to_nchar(job) job,
mgr mgr,
hiredate hiredate,
sal sal,
deptno deptno,
 to_nchar(org) org');
```

The way to change CLOB to NCLOB columns using redefinition is very similar. The difference is that, in the above redefinition statement, the corresponding explicit conversion function should be TO_NCLOB, instead of TO_NCHAR.

4. Create any triggers, indexes, grants and constraints on the interim table. Any referential constraints involving the interim table (that is, the interim table is either a parent or a child table of the referential constraint) must be created disabled. Until the redefinition process is either completed or aborted, any trigger defined on the interim table will not execute.

5. Optionally, synchronize the interim table `int_emp`. Before `FINISH_REDEF_TABLE` is called, if a large number of DML operations have been applied on the original table, the interim table should be periodically synchronized with the original table by executing the SYNC_INTERIM_TABLE procedure. This reduces the time taken by `FINISH_REDEF_TABLE()` to complete the redefinition process.

```
EXECUTE DBMS_REDEFINITION.SYNC_INTERIM_TABLE('scott', 'emp', 'int_emp');
```

6. Execute the `DBMS_REDEFINITION.FINISH_REDEF_TABLE()` procedure to complete the redefinition of the table.

EXECUTE DBMS_REDEFINITION.FINISH_REDEF_TABLE(‘scott’, ‘emp’, ‘int_emp’);

As a result of this procedure, the following are applied to the original table.

- The original table is redefined such that it has all the attributes, indexes, constraints, grants and triggers of the interim table.
- The referential constraints involving the interim table now involve the post redefined table and are enabled.

7. Drop the interim table

DROP TABLE int_emp;

The end result of the above redefinition procedures is:

- The original table is migrated to Unicode columns.
- The triggers, grants, indexes and constraints defined on the interim table after `START_REDEF_TABLE()` and before `FINISH_REDEF_TABLE()` are now defined on the post-redefined table. Any referential constraints involving the interim table before the redefinition process was finished now involve the post-redefinition table and are enabled.
- Any indexes, triggers, grants and constraints defined on the original table (prior to redefinition) are transferred to the interim table and are dropped when the user drops the interim table. Any referential constraints involving the original table before the redefinition now involve the interim table and are disabled.
- Any PL/SQL procedures and cursors defined on the original table (prior to redefinition) are invalidated. They are automatically revalidated (this revalidation can fail if the shape of the table was changed as a result of the redefinition process) whenever they are used next.

NOTE: There are a number of restrictions that apply to online table redefinition. Please refer to the *Database Administrator's Guide* for detailed information.

Example for CLOB Schema Migration

The following is an example of migrating CLOB columns to NCLOB columns. Please note the difference in the column mapping specification. TO_NCLOB is used to map the original CLOB column to the NCLOB column.

```
CREATE TABLE lb (anum NUMBER PRIMARY KEY, lb CLOB);
CREATE TABLE int_lob (n1 NUMBER PRIMARY KEY, nlb NCLOB);

EXECUTE DBMS_REDEFINITION.CAN_REDEF_TABLE('Scott', 'lb');
EXECUTE DBMS_REDEFINITION.START_REDEF_TABLE('Scott', 'lb', 'int_lob', 'anum n1,
TO_NCLOB(lb) nlb');
EXECUTE DBMS_REDEFINITION.SYNC_INTERIM_TABLE('Scott', 'lb', 'int_lob');
EXECUTE DBMS_REDEFINITION.FINISH_REDEF_TABLE('Scott', 'lb', 'int_lob');

DROP TABLE int_lob;
```

By using the online table redefinition, the down time of the table is reduced. This assumes that there are minimal DML operations on the original table before the redefinition is finally completed. The speed for DML operations on the original table before redefinition process is completed is a little slower.

Comparison Between “Alter table” and “On-line Table Redefinition”

Both the ALTER TABLE command and on-line table redefinition can change a column’s definition. Each one has its advantage in certain scenarios.

ALTER TABLE command:

a. Easy to use.
b. Fewer restrictions.

On-line table redefinition:

a) Better performance. On-line table redefinition is generally faster than the ALTER TABLE command.
b) Can migrate several columns at one time.
c) Table available for DDL during most of the migration process.
d) Avoids table fragmentation which leads to space saving and faster access.
e) Works for CLOB to NCLOB migration.

Step 5. Post Migration Tasks

Some actions need to be taken to recover migrated database schema to its original state. Also, the result of schema migration may have other effects that indirectly affect associated applications. The user should pay attention to these effects and take appropriate actions to prevent them.

Index: When the table columns are changed from SQL CHAR types to Unicode datatypes by the ALTER TABLE MODIFY command, the index built on top of it will be changed automatically by the database system. However, this also slows down performance for the ALTER TABLE command. If indexes are dropped before the ALTER TABLE command is issued, they should be recreated after migration.

Constraints that are disabled before migration need to be re-enabled after migration.

Triggers that are disabled before migration should be enabled again after migration.

Replication: If the columns that are migrated to Unicode types are replicated across several sites, the data changes due to migration will be propagated to different sites either synchronously or asynchronously depending on the replication definition.

Binary Order: The migration from SQL CHAR column to Unicode datatype involves character set conversion if database and NCHAR have different character sets. The binary order of the same data in a different character set encoding may be different, potentially effecting applications that rely on this order.

Application Migration to Unicode Datatypes

Oracle provides a comprehensive set of database access interfaces that can be used by applications in a middle tier or client. All the major access interfaces support Unicode datatypes at various levels. This chapter talks about how to migrate existing applications to support Unicode data types. We will focus on three popular database access interfaces and environments: PL/SQL, OCI and JDBC.
The implicit conversion between SQL NCHAR types and other Oracle datatypes including the SQL CHAR types significantly reduces the workload to migrate to Unicode datatypes. If the application relies on implicit conversion for handling the operations between SQL CHAR and SQL NCHAR, only minor changes are needed to make the application adapt to the newly added SQL NCHAR columns or tables. If all the Unicode columns are converted from SQL CHAR columns and no new Unicode column is added to existing schema, theoretically no change is necessary to make the existing application run with Unicode datatypes for the existing data. However, when new Unicode data is added that may not be covered by the database character set, then application migration is required to avoid possible data loss. Also, in order to make the application run more efficiently with Unicode data, some practice of code modifications will benefit performance as well. Here we will give some guidelines for migration to Unicode datatypes. They can make migration easier and result in better performance.

PL/SQL Application

User should pay attention to several things during migration for PL/SQL applications. They are discussed in the following subsections.

PL/SQL Variables

There are several types of data or PL/SQL variables that will be processed in the PL/SQL environment. One type of variable is used to hold and process data that interact with database table columns. Such variables include the following categories:

a. To hold data retrieved from database or to be inserted into database columns.

b. To interact with database table columns in operations such as comparison, concatenation, and SQL functions.

c. To interact with variables in category a. and b.

PL/SQL variables in the above categories should be synchronized with the database columns that they interact with. For a database that is fully migrated to Unicode datatypes, these variables should be defined as SQL NCHAR types. For a database that is partially migrated to Unicode, the user should be careful and be aware of the datatypes of table columns. %TYPE and %ROWTYPE syntax can be used to make such PL/SQL variables synchronous with database columns. Here is an example of how to use this syntax:

```
DECLARE
my_ename NVARCHAR2(10);
my_job emp.job%TYPE;
BEGIN
SELECT ename, job INTO my_ename, my_job FROM emp where ename='SCOTT';
END;
```

The synchronization between PL/SQL variables and database columns or between PL/SQL variables is not mandatory since implicit conversion between CHAR and NCHAR can automatically convert them if necessary. However, we recommend that the user always follow the above guideline to avoid the following consequences

1. Data loss. When data in SQL NCHAR columns are selected into SQL CHAR PL/SQL variables, or SQL NCHAR variables are inserted into SQL CHAR database columns, or SQL NCHAR variables are assigned to SQL CHAR variables, data loss could happen if the database character set is only a subset of the NCHAR character set.
2. Performance overhead. The conversion between SQL CHAR and SQL NCHAR can introduce some performance overhead. Synchronization can eliminate such kind of overhead.

Another category of PL/SQL variables are those that are related to metadata, (such as user name, password, tables, views, procedures, functions, types, and columns) and SQL statements. They are stored and processed in the database as SQL CHAR types. Therefore, there is no need to migrate PL/SQL variables in this category. Migrating them to SQL NCHAR types will introduce unnecessary conversions and performance overhead since they will be eventually converted back to SQL CHAR types using the native database encoding.

EXAMPLE:

```
DECLARE
sql_command VARCHAR2(100);
……
BEGIN
sql_command := 'INSERT INTO ' || tabName || ' (col1, col2, col3 ) VALUES( :var1, :var2, :var3 )';
dbms_sql.parse( sqlCursor, sqlCommand, dbms_sql.v7 );
dbms_sql.bind_variable( sqlCursor, ':var1', varVal );
……
END;
```

The above example shows that there is no need to migrate the ***SQL command*** variable to a Unicode datatype.

Some PL/SQL variables do not interact with the database either directly or indirectly. These variables can be defined as either SQL NCHAR types or SQL CHAR types depending on whether the data is defined in the database character set.

```
DECLARE
var1 VARCHAR2(10);
var2 VARCHAR2(20);
… …
BEGIN
……
var2 := var1;
… …
END;
```

PL/SQL String Literal

To indicate whether a string literal is a SQL NCHAR type, character 'N' can be placed in front of the string literal. The 'N' prefix is not mandatory. User could omit the 'N' and rely on implicit conversion to convert the literal into a SQL NCHAR type if necessary. The string literal embedded in PL/SQL programs may or may not interact with SQL NCHAR types. For string literals that interact with SQL NCHAR variables, it is recommended to prefix the letter 'N' before the single quote of a string literal to indicate that the literal is NCHAR type. The result is that the literal is converted to the NCHAR character set encoding and treated the same as SQL NCHAR data at compile time. Otherwise the literal is treated as a CHAR type at compile time and converted to NCHAR at run time each time the statement is executed. This can save conversion overhead if the literal is in a loop or the containing procedure is invoked multiple times. For example:

```
DECLARE
name NVARCHAR(2000);
BEGIN
  FOR i IN 1 .. 2000 LOOP
     name := name || N‘ ‘;
  END LOOP;
END;
```

PL/SQL String Comparison

For comparison operations between SQL CHAR and SQL NCHAR types in SQL queries, implicit conversion always converts from SQL CHAR to SQL NCHAR type to avoid data loss. The comparison could be between two database columns or between a database column and a literal or results from a SQL function or query. If data loss will not happen, the user can apply an explicit conversion function to convert the side with the lesser number of conversions. This is true for all such SQL queries that can be embedded in PL/SQL, OCI or any other development environment.
Example:

```
tab_a  (col_a VARCHA2(100), ……) has 1 million rows
table tab_b (col_b NVARCHAR2(100), ……) has 100 rows
select col_a from tab_a, tab_b where tab_a.col_a = TO_CHAR(tab_b.col_b);
```

The explicit conversion converts *col_b* from Unicode type to VARCHAR2 type, resulting in only 100 conversions. If relying on implicit conversion, *col_a* will be converted to Unicode type, resulting in 1 million conversions.

PL/SQL Procedures/Functions

PL/SQL procedures/functions that only support SQL CHAR types need to be migrated to support both SQL NCHAR and SQL CHAR types. Without migrating to support Unicode types, Unicode data passed to the PL/SQL procedures as SQL NCHAR types will be implicitly converted to SQL CHAR types first. Data loss may happen if some of the Unicode data cannot be mapped to the database character set. There are three ways to support SQL NCHAR types in PL/SQL procedures.

- Convert the arguments and return types of the PL/SQL procedures/functions as well as the local variables from SQL CHAR to SQL NCHAR types. The following is an example.

```
FUNCTION get_str(cur IN NVARCHAR2,
                 old IN NVARCHAR2,
                new IN NVARCHAR2)
RETURN NVARCHAR2 IS
  local_var NVARCHAR2(10);
BEGIN
local_var := cur;
……
END;
```

Even though the above PL/SQL procedure is defined using SQL NCHAR types, it can work for both SQL CHAR and SQL NCHAR datatypes. At run time, if arguments of SQL CHAR types are passed in, implicit conversion can automatically convert SQL CHAR data to SQL NCHAR types. As a result, some performance overhead is introduced.

- ANY_CS can be used to declare the datatype of a parameter to a procedure/function for CHAR/VARCHAR2/CLOB type. It indicates that the parameter inherits its character set (which determines whether it’s a SQL CHAR or SQL NCHAR type) from the actual argument value at run time. %CHARSET can copy the character set of a parameter or variable declared with a character set name ANY_CS. It can be applied to the name of an earlier parameter in the same parameter list to indicate that the parameter being declared must be passed with the same character set as the earlier parameter. It can also be used to declare local variables to indicate that this variable is in the same character set as the parameter passed in or the same character set as another character type variable. By using ANY_CS and %CHARSET, one PL/SQL procedure/function can support both SQL CHAR and SQL NCHAR types. At the same time, variables can be guaranteed to be the same character set as each other. This reduces the number of character set conversions for operations between SQL CHAR and SQL NCHAR types.

```
FUNCTION get_str(cur IN VARCHAR2 CHARACTER SET ANY_CS,
                  old IN VARCHAR2 CHARACTER SET cur%CHARSET,
                 new IN VARCHAR2 CHARACTER SET cur%CHARSET)
RETURN VARCHAR2 CHARACTER SET cur%CHARSET IS
   local_var VARCHAR2(10) CHARACTER SET cur%CHARSET;
BEGIN
……
END;
```

- Define a separate procedure/function for SQL NCHAR types. This can be used if there are only a small number of procedure/functions that need to be redefined.

Length Semantics Change

Character length semantics is the only length semantics of the Unicode datatypes. PL/SQL applications need to make adjustments for this semantics change. For the PL/SQL variables that are made synchronous with database columns, the length semantics is made synchronous at the same time. This means that if the variable is determined to be SQL NCHAR types based on %TYPE or %ROWTYPE syntax, its length semantics is automatically character length semantics. The parameter list in the PL/SQL procedure specification does not have any length constraints. So the length semantics is not an issue for PL/SQL procedures. At run time, when SQL NCHAR data is passed down to the procedure, the parameter's length semantics is implicitly character length semantics.

String Functions

In PL/SQL applications, some length related SQL or PL/SQL internal functions are used for various purposes. Some of these functions are based on number of bytes, such as LENGTHB, SUBSTRB, and INSTRB. Some other functions are based on number of characters such as LENGTH, SUBSTR, INSTR or LENGTH2, SUBSTR2 and INSTR2 (refer to the *SQL Reference*). After the PL/SQL variables and procedures are migrated to Unicode datatypes, the internal functions that operate on them should also be migrated at the same time. These internal functions are used for various purposes. They need to be examined and migrated properly. Here is a brief summary of their usage.

I. In most cases, the character-based functions are used for both single-byte and multibyte data. They already understand character length semantics, so there is no need to do anything for them.

II. The byte-based functions are used in the following scenarios.

- Check the existence of multibyte characters by checking if the byte length is equal to character length.

```
DECLARE
context NVARCHAR2(10);
……
BEGIN
……
if length(context) != lengthb(context) then…
END;
```

Since *context* is now Unicode type, this checking always fails for AL16UTF16 encoding because all characters must be at least two bytes long. Such conditional checking should be removed.

- Check if the input string is less than a limit that is number of bytes. The SQL functions should be changed to character-based, and the meaning of the corresponding limit is changed to number of characters.

```
If lengthb(context) > 10 then raise context_too_long;
```
change to:
```
If length(context) > 10 then raise context_too_long;
```

- Use of byte length as loop boundaries to process every byte in the string. The processing can be changed to character by character and the byte-based function should be changed to a character-based function.

```
DECLARE
buffer NVARCHAR2(200);
val NUMBER;
……
BEGIN
  FOR i IN 1 .. LENGTHB(buffer) LOOP
     val := ASCII(SUBSTRB(buffer, i, 1));
     ...
  END LOOP;
END;
```

The LOOP statement should be changed to:
```
FOR i IN 1 .. LENGTH(buffer) LOOP
    val := ASCII(SUBSTR(buffer, i, 1));
```

The original PL/SQL program only works for a single-byte character set. After the application is migrated to support Unicode datatypes, changes must be made to make it work properly. Single byte in AL16UTF16 encoding does not carry any meaning. Only two-byte code units are meaningful. The function should be changed to be same length semantics as the variables. Therefore LENGTHB and SUBSTRB should be changed to LENGTH and SUBSTR.

- The byte position of a string has special meaning. The byte position can be changed to a character position.

```
DECLARE
colon NUMBER;
doc_info NVARCHAR2(200);
node_id NUMBER;
BEGIN
colon := INSTRB(doc_info, ':');
```

```
node_id := to_number(SUBSTRB(doc_info, 1, colon-1));
END;
```

The last two lines should be changed to:

```
colon := INSTR(doc_info, ':');
node_id := to_number(SUBSTR(doc_info, 1, colon-1));
```

INSTRB and SUBSTRB need to be changed to INSTR and SUBSTR because the operand *doc_info* is defined as NVARCHAR2, which always has character semantics.

- Make sure a string fits in another variable. The length semantics of the function should be made consistent with the destination variable definition.

```
DECLARE
MSG NVARCHAR2(2000);
……
BEGIN
MSG := substrb(MSG||' ('||TOK_NAM||'='||TOK_VAL||')',1,2000);
……
END;
```

change to:

```
MSG := substr(MSG||' ('||TOK_NAM||'='||TOK_VAL||')',1,2000);
```

SUBSTRB should be changed to SUBSTR because MSG is defined as character length semantics. SUBSTRB can result in truncation in the middle of a character.

Based on the above usage scenarios, the conclusion is that the byte-based functions operating on SQL NCHAR types should be changed to character-based functions.

III. SQL Functions of different length semantics.

Oracle provides length related SQL functions to meet varying application environments. They are distinguished by the character or number attached to the function's name, such as SUBSTR, SUBSTRB, SUBSTR2, SUBSTR4 and SUBSTRC. The following are the major differences between them and also apply to LENGTH, LENGTHB, LENGTH2, LENGTH4, LENGTHC and INSTR, INSTRB, INSTR2, INSTR4, INSTRC.

`SUBSTR` calculates lengths in character units as defined by the character set of the datatypes. For example, AL32UTF8 is calculated in UCS4 code units. UTF8 and AL16UTF16 are in UCS2 code units. Therefore, supplementary characters will be counted as one character in AL32UTF8 and two characters in AL16UTF16. Also, this means that SUBSTR may use different character units for VARCHAR and NVARCHAR since the character sets will likely be different. If consistency is important it may be better to use SUBSTR2 or SUBSTR4 to force all semantic calculations to UCS2 or UCS4 respectively.

`SUBSTRB` calculates length in bytes.

`SUBSTR2` calculates lengths in UCS2 code units, which is compliant with Java strings and Windows client environments. Supplementary characters count as two code units.

`SUBSTR4` calculates lengths in UCS4 code units. Supplementary characters count as one code unit.

SUBSTRC calculates lengths in Unicode complete characters. Supplementary characters and composite characters are counted as one character.

CLOB to NCLOB Migration

All the above discussions apply to migration from CLOB to NCLOB as well. Implicit conversion is supported between CLOB and NCLOB in Oracle Database 10g. However in Oracle9*i* and in general best practice, TO_NCLOB or TO_CLOB SQL functions should use explicit operations between them. Any operation attempting to rely on implicit conversion will result in a compilation error in Oracle9*i*. In the following example, TO_NCLOB is used.

```
DECLARE
alb CLOB;
nlb NCLOB;
BEGIN
  nlb := CONCAT(nlb, TO_NCLOB(alb));
 ……
   END;
```

OCI Application

There are no separate external OCI types for SQL NCHAR types. The external OCI types for SQL_NCHAR are the same as those for SQL CHAR types. OCI supports Unicode datatypes through the OCI attribute OCI_ATTR_CHARSET_FORM for bind/define buffers. The values for this parameter are SQLCS_IMPLICIT (indicates database character set ID) and SQLCS_NCHAR (indicates national character set ID). When OCI_ATTR_CHARSET_FORM is set to SQLCS_NCHAR, data will be converted to/from NCHAR character set on server from/to NCHAR character set setting on client. When OCI_ATTR_CHARSET_FORM is set to SQLCS_CHAR, the data in define/bind buffer is converted from/to the database character set on client side. Below, we list each OCI usage scenario and describe how OCI applications should be migrated in each case.

- OCI in a totally Unicode application: We recommend using OCIEnvNlsCreate to specify the SQL CHAR and SQL NCHAR character set as UTF-16. When *OCI_UTF16ID* is specified for both SQL CHAR and SQL NCHAR all of the character data in UTF-16 encoding regardless of the NLS_LANG setting. This includes all of the metadata as well as user data. When OCI application retrieves data from columns of SQL NCHAR into a define buffer, or when it inserts data into SQL NCHAR columns from bind buffers, OCIAttrSet should be called to set OCI_ATTR_CHARSET_FORM attributes associated with the define/bind buffers. Without setting the OCI_ATTR_CHARSET_FORM attribute to be SQLCS_NCHAR, all the data in bind buffers are converted to the database character set by OCI before they are sent to the database server. If the database character set is only a subset of the NCHAR character set, data loss might happen when the eventual destination of the data are columns of the SQL NCHAR datatypes. Such data loss can be avoided by setting OCI_ATTR_CHARSET_FORM attribute to SQLCS_NCHAR.

OCIEnvNlsCreate for Unicode programming:

```
OCIEnvNlsCreate(envhpp, OCI_DEFAULT,  …, OCI_UTF16ID ,OCI_UTF16ID);
```

OCI_ATTR_CHARSET_FORM attribute can be set by the following function.

```
ub1 charsetfm = SQLCS_NCHAR;
OCIAttrSet((dvoid *)bindp, (ub4) OCI_HTYPE_BIND, (dvoid *)&charsetfm, (ub4) 0,
(ub4) OCI_ATTR_CHARSET_FORM, errhp));
```

- OCI in other applications: There are different types of OCI applications that work with data from SQL NCHAR columns. Such applications may be an end user application involving some user interaction, or it can be a middle tier application such as a gateway that only lets data pass through and does not do any data processing. It can also be an application that does data processing. In each case, we recommend that the user set the OCI_ATTR_CHARSET_FORM attribute to SQLCS_NCHAR if the column which the client buffer is associated with is SQL NCHAR type and set OCI_ATTR_CHARSET_FORM attribute to SQLCS_IMPLICIT if the column is SQL CHAR type (this is the default case, so user does not need to set anything). The reasons are twofold.

    – This avoids data loss if database character set is only a subset of NCHAR character set.
    – Once this attribute is set to be the same character set form as the database column, data in define/bind buffer will be converted to the proper character set on client rather than doing the conversion on the server side. This can reduce the server workload.

- CLOB to NCLOB migration in OCI: The migration from CLOB to NCLOB is similar to the migration from other SQL CHAR to SQL NCHAR datatypes. Implicit conversion is not supported between CLOB and NCLOB in Oracle9*i*. The user must set the OCI_ATTR_CHARSET_FORM attribute correctly according to the type of the target table column. Otherwise, an error will be returned. Note in Oracle Database 10g onward implicit conversion between CLOB and NCLOB is supported so an error would not be returned. Since handling the NCLOB type is much different from NCHAR/NVARCHAR2, here we list an example to demonstrate how to code for a NCLOB application.

The following example is based on table *FOO (A INT, C NCLOB).*

```
char *insstmt = (char *)"INSERT INTO FOO (A, C) VALUES (1, :1);"
char *selstmt = (char *)"SELECT C FROM FOO WHERE A = 1";
ub1   buf[MAXBUFLEN];
ub1   *rbuf;
ub4   blen = 0;
ub4   loblen = 0;
OCILobLocator *clob;

if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &clob,
                  (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
{
  return OCI_ERROR;
}

if (OCIStmtPrepare(stmthp, errhp, insstmt, (ub4)strlen((char *)insstmt),
               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
{
  return OCI_ERROR;
}

memset((void *) buf, (int) 'A', (size_t) MAXBUFLEN);

if (OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
                (dvoid *) buf, (sb4) inputlen,
                SQLT_CHR,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
{
  return OCI_ERROR;
}

if (OCIAttrSet((dvoid *) bindp, (ub4) OCI_HTYPE_BIND,
                         (dvoid *) SQLCS_NCHAR, (ub4) 0,
                         (ub4) OCI_ATTR_CHARSET_FORM, errhp))
{
   return OCI_ERROR;
}

retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                         (OCISnapshot*) 0, (OCISnapshot*) 0,
                         (ub4) OCI_DEFAULT);

/** The following statements may not be needed for this example.
    But just in case the character set and id are needed in other scenarios.  **/

if (OCILobCharSetId(envhp, errhp, clob, &csid))
{
  return OCI_ERROR;
}

if (OCILobCharSetForm(envhp, errhp, clob, &csform))
{
  DISCARD printf("FAILED: OCILobCharSetForm()\n");
  report_error(errhp);
}

if (OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char *)selstmt),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
{
  return OCI_ERROR;
}

if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                  (OCISnapshot*) 0, (OCISnapshot*) 0,
                  (ub4) OCI_DEFAULT))
{
  return OCI_ERROR;
}

if (OCIDefineByPos(stmthp, &dfnhp, errhp, (ub4) 1,
                  (dvoid *) &clob, (sb4) -1,
                   SQLT_CLOB, (dvoid *) 0,
                  (ub2 *) 0,(ub2 *) 0, (ub4) OCI_DEFAULT))
{
  return OCI_ERROR;
}

if (OCIAttrSet((dvoid *) bindp, (ub4) OCI_HTYPE_BIND,
                         (dvoid *) &csform, (ub4) 0,
                         (ub4) OCI_ATTR_CHARSET_FORM, errhp))
```

```
{
  return OCI_ERROR;
}

if (OCIStmtFetch(stmthp, errhp, (ub4) 1, (ub2) OCI_FETCH_NEXT,
              (ub4) OCI_DEFAULT))
{
 return OCI_ERROR;
}

/* OCILobGetLength returns the lob length in number of characters */
if (OCILobGetLength(svchp, errhp, locator, &loblen))
{
 return OCI_ERROR;
}

/** Maximum number of bytes per character is 4.  The character length multiplied
     by the maximum number of bytes per character can guarantee the retrieved data
    fits  in the buffer  **/

#define MAX_BYTE_LEN 4
blen = loblen*MAX_BYTE_LEN;

rbuf = (ub1 *)malloc(blen);
memset((void *) rbuf, (int) '\0', (size_t) blen);

if (OCILobRead(svchp, errhp, locator, &amtp, (ub4) pos, (dvoid *) rbuf,
             (ub4) loblen, (dvoid *)0,
             (sb4 (*)(dvoid *, CONST dvoid *, ub4, ub1)) 0,
             (ub2) 0, (ub1) SQLCS_NCHAR ))
{
 report_error(errhp);
}
else
{
 if (memcmp((const void *) rbuf, (const void *) rbuf2, (size_t)MAXBUFLEN))
   DISCARD printf("FAILED: OCILobRead(); buffers differ\n");
 else
   DISCARD printf("PASSED: OCILobRead(); buffers equal\n");
}
```

JDBC Application

In JDBC, there are no separate, corresponding datatypes or classes defined for SQL NCHAR types. It uses the same classes and methods to access SQL NCHAR datatypes as SQL CHAR datatypes. So the usage of SQL NCHAR datatypes is similar to that of the SQL CHAR datatypes. The following are the major differences in dealing with SQL CHAR and SQL NCHAR types.

a. When a JDBC program binds data, it must call setFormOfUse() method to specify that the data is bound for SQL NCHAR datatypes. This is similar to the OCIAttrSet function that can set the character set form attribute. There are two valid values for the character set form: FORM_CHAR and FORM_NCHAR. FORM_CHAR is the default value. If FORM_NCHAR is set as the form of use, JDBC driver will represent the data in the national character set of the server internally. FORM_NCHAR should be used for all buffers corresponding to SQL NCHAR datatypes on the server. The following code demonstrates how to access SQL NCHAR data.

```
int empno = 12345;
String ename = "\uFF2A\uFF4F\uFF45";
String job = “Engineer”;
oracle.jdbc.OraclePreparedStatement pstmt = (oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("INSERT  INTO emp (empno, ename, job) VALUES(?, ?, ?)");

pstmt.setFormOfUse(2, FORM_NCHAR);
pstmt.setFormOfUse(3, FORM_NCHAR);

pstmt.setInt(1, 1);
pstmt.setString(2, ename);
pstmt.setString(3, job);
pstmt.execute();
pstmt.close();
```

NOTE: The *setFormOfUse* must be called before *setString* to get proper results.

b. The oracle.sql.CHAR class is designed for both SQL CHAR and SQL NCHAR datatypes embedded in an Oracle object type. It is used by Oracle JDBC in handling and converting character data. One of the key attributes of a CHAR object is the character set that defines the encoding of the character data in the object. It must be specified when a CHAR object is created. When a CHAR object is created for SQL NCHAR datatypes, the database national character set of the server should be used. If it is for SQL CHAR datatypes, one of US7ASCII, WE8ISO8859P1 or UTF8 should be used depending on the database character set. Please refer to the JDBC manual for detailed information about CHAR objects. The following is an example to create CHAR object for SQL NCHAR datatypes.

```
int oracleId = CharacterSet.AL16UTF16_CHARSET;  // Character set ID for
AL16UTF16
...
CharacterSet mycharset = CharacterSet.make(oracleId);
String mystring = "\uFFA0";
...
CHAR mychar = new CHAR(mystring, mycharset);
```

c. CLOB to NCLOB migration for JDBC applications

Similarly, the user needs to set the character set form attribute to FORM_NCHAR for the data that is targeted toward NCLOB columns. Implicit conversion is not supported between CLOB and NCLOB in Oracle9*i*. Without setting this attribute to be the same character set form as the target CLOB/NCLOB column, an error will be returned in Oracle9*i*. Note in Oracle Database 10g onward implicit conversion between CLOB and NCLOB is supported so an error would not be returned, but the implicit conversion may have a performance impact. The following is an example of how to code against NCLOB columns in JDBC applications.

The following example is based on table definition
clob_table (v2 VARCHAR2 (30), ncb NCLOB).

```
Connection conn;
try {
    OraclePreparedStatement  pstmt =
      (oracle.jdbc.OraclePreparedStatement) conn.prepareStatement
       ("insert into clob_table values (?, ?)");
```

```
  pstmt.setFormOfUse(2, OraclePreparedStatement.FORM_NCHAR);

  pstmt.setString (1, "one");
  pstmt.setString (2, "\uFF10\uFF11\uFF12\uFF13\uFF14");
  pstmt.execute ();

}

ResultSet rset = stmt.executeQuery ("select * from clob_table where v2
                                     = 'one' for update");
if (rset.next ())
{
  oracle.jdbc2.Clob clob = ((OracleResultSet)rset).getClob (2);
  show("getLength() = "+clob.length());

  String str = clob.getSubString(1,5);
  String data = "\uFF41\uFF42\uFF43\uFF44\uFF45";
  ((CLOB)clob).putString(1, data);
}
```

To summarize for JDBC application migration, because the JAVA internal string encoding is UTF-16, not much needs to be done to support Unicode datatypes besides the above two scenarios. When variables are bound to SQL NCHAR datatypes, the form of use attribute needs to be set to FORM_NCHAR. Also when a CHAR object is created, the database NCHAR character set should be supplied.

Summary

This paper discussed migration steps to support Unicode datatypes for both the database and applications. Unicode datatypes allow the user to do incremental migration to Unicode. The issues such as the criteria to pick Unicode character set, methods to migrate to Unicode schema and some important things for user to note are discussed in the database migration.

Implicit conversion between SQL CHAR and SQL NCHAR types significantly reduces the workload to migrate applications to Unicode datatypes. However, to avoid data loss and achieve the best performance for the migrated applications, users need to follow some guidelines. Based on the discussion in the application migration, when SQL NCHAR columns are accessed from client or middle tire applications, special settings should be made to access them as SQL NCHAR types by setting character form attribute or synchronizing with the database columns using %TYPE or %ROWTYPE syntax. This avoids data loss. The key to achieving the best performance is to minimize the operations between SQL NCHAR and SQL CHAR types as much as possible. Application developers should try to avoid the scenario where operations between mixed datatypes are required. When operations are not avoidable, the user can follow the above guidelines to reduce the number of conversions.

ORACLE

Migration to Unicode Datatypes for Multilingual Databases and Applications
January 2005
Author: Gary Chen
Contributing Authors: Barry Trute

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com