Informix Unicode DataBlade Module

User's Guide

Version 1.0
August 1998
Part No. 000-5211

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Table of Contents

Introduction

In This Introduction

This chapter introduces the *Informix Unicode DataBlade Module User's Guide*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout the book.

About This Manual

This manual contains information to help you use the Informix Unicode DataBlade module with your Informix database server. A DataBlade module adds functionality to your database server by providing custom data types and supporting routines. The Unicode DataBlade module enables customers to store Unicode data in database tables and to manipulate and retrieve that data using SQL.

This section discusses the organization of the manual, the intended audience, and the software products you must have to use the Unicode DataBlade module.

Organization of This Manual

The *Informix Unicode DataBlade Module User's Guide* contains an overview chapter and several reference chapters. At the end of the book, there is a glossary.

This manual includes the following chapters:

- This introduction provides an overview of the manual, describes the documentation conventions used, and explains the generic style of this documentation.

- Chapter 1, "Overview," introduces the Unicode DataBlade module and provides general information about the product.
- Chapter 2, "Unicode Columns," describes the UNIvarchar data type. It also includes some relevant information about the LVARCHAR data type.
- Chapter 3, "Casting and Unicode," provides information about the implicit and explicit casts and code set conversions that are necessary when using the UNIvarchar data type. It also includes some examples.
- Chapter 4, "Functions," contains reference pages for routines used with this DataBlade module.
- Chapter 5, "Module-Specific Syntax," explains what you need to know to use common SQL statements with this DataBlade module. In particular, it discusses the CREATE TABLE statement and the SELECT statement.
- Chapter 6, "The gl_conv Utility," describes the feature that enables users to store binary data in UNIvarchar columns.
- Appendix A, "Sample Tables," contains the CREATE TABLE and INSERT statements for the tables used in the examples in this book.

A glossary of terms used with this DataBlade module and some common database terms follows the chapters, and a comprehensive index directs you to areas of particular interest.

Types of Users

This manual is written for:

- customers who want to create a database in which to store data in many natural languages.
- customers who have Unicode data from a source that they want to use with Informix databases. Typically this would be a customer migrating from another database manufacturer, but it could be a customer migrating Unicode data from a third-party application, such as a spreadsheet.

Software Dependencies

To use this product, you must be using Informix Dynamic Server with Universal Data Option and the Global Language Support (GLS) feature.

Features of This Product

The Unicode DataBlade module extends the functionality of your Informix database server by providing:

- an opaque data type to handle Unicode data: UNIvarchar.
- custom routines to support the new data type.
- implicit and explicit casts for UNIvarchar data.
- a utility to convert binary data to Unicode.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Function syntax conventions
- Sample-code conventions

Typographical Conventions

This manual uses the following standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax statements, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
`monospace`	Information that the product displays on the screen and information that you enter appear in a monospace typeface.

Tip: *The text and many of the examples in this manual show function and data type names in mixed lettercasing (uppercase and lowercase). Because your Informix database server is case insensitive, you do not need to enter function names exactly as shown: you can use uppercase letters, lowercase letters, or any combination of the two.*

Icon Conventions

Throughout this document, you will find text identified by icons in the margin. These icons identify three types of information, as described in the following table. This information is always displayed in italics.

Icon	Label	Description
⚠	***Warning:***	Identifies paragraphs that contain vital instructions, cautions, or critical information
⇨	***Important:***	Identifies paragraphs that contain significant information about the feature or operation that is being described
💡	***Tip:***	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Function Syntax Conventions

This guide uses the following conventions to specify syntax:

- Square brackets ([ ]) surround optional items.
- Curly brackets ({ }) surround items that can be repeated zero or more times.
- A vertical line (|) separates alternatives.
- Function parameters are italicized; arguments that must be specified as shown are not italicized.
- Repeatable segments of a function are indicated with ellipses. For example, the syntax: [(id_num [,id_num...])] represents an optional list of any number of ID values, separated by commas.

These syntax conventions are used in Chapter 4 and Chapter 6 of this guide.

Printed Documentation

The following related Informix documents complement the information in this manual set:

- The *Getting Started* guide for your database server describes the architecture and features of your database server and offers instructions for installing and configuring it.
- Whoever installs your Informix products should refer to the *Installation Guide* for your particular platform and release to ensure that your Informix product is properly set up before you begin to work with it. The installation guides include a matrix that depicts possible client / server configurations.
- Before you can use the Unicode DataBlade module, you must install and configure your database server. The *Administrator's Guide* for your database server provides information about how to configure your server to interact with DataBlade modules.
- Installation instructions for the Unicode DataBlade module are provided in the hard copy *Read Me First* sheet for DataBlade modules that is packaged with this product. Once you have installed the DataBlade module, you must use BladeManager to register it into the database where the DataBlade module will be used. See the *DataBlade Module Installation and Registration Guide* for details on registering DataBlade modules.
- To learn about the Informix version of Structured Query Language (SQL), read the *Informix Guide to SQL: Tutorial*. It provides a tutorial on SQL as it is implemented by Informix products. It also describes the fundamental ideas and terminology for planning and implementing a relational database.
- Reference pages for SQL statements used with the Unicode DataBlade module are in the *Informix Guide to SQL: Syntax*. (Module-specific adjustments to this syntax are described in Chapter 5 of this manual.)
- A companion volume to the *Tutorial* and *Syntax*, the *Informix Guide to SQL: Reference* includes details of the Informix system catalog tables, describes Informix and common environment variables that you should set, and describes the column data types that Informix database servers support.

Tip: *This book also contains a glossary of server-related terms.*

- The *Informix Guide to GLS Functionality* contains important information about character sets, code sets, multibyte data, localization, and working with multiple natural languages.
- The *Informix ESQL/C Programmer's Manual* is essential for anyone writing programs using ESQL/C.
- *Informix Error Messages* is useful if you do not want to look up your error messages on-line.

Consult your Informix representative to find out what recent documentation is available for your database server configuration.

On-Line Documentation

Several different types of on-line documentation are available:

- On-line manuals
- On-line help
- On-line error messages
- Documentation notes, release notes, and machine notes

On-Line Manuals

All the Unicode DataBlade module manuals are provided on a CD so that you can view and search for information on-line.

Documentation Notes, Release Notes, Machine Notes

In addition to the Informix set of manuals, the following on-line files supplement the information in this manual.

On-Line File	Purpose
Documentation notes	Describes features not covered in the manuals or modified since publication.
Release notes	Describes any special actions required to configure and use the DataBlade module on your computer. Additionally, this file contains information about any known problems and their workarounds.
Machine notes	Describes platform-specific information regarding the release.

Please examine these files because they contain vital information about application and performance issues.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Please include the following information:

- The name and version of the manual you are using
- Any comments you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

doc@informix.com

We appreciate your feedback.

Chapter

Overview

1

In This Chapter

This book provides instructions on how to use the Informix Unicode DataBlade module.

What is the Unicode DataBlade Module?

The Unicode DataBlade module enables a user to store data in Unicode format and to access and manipulate data that has been stored in Unicode format.

The Unicode DataBlade module allows you to use existing SQL statements to store, access, and manipulate native Unicode data in the same way you manipulate any variable character data. If you are using ESQL/C, you can retrieve the actual Unicode data. If you are using DB-Access or another dynamic SQL tool, your data will be the hexadecimal ASCII representation of Unicode, returned as an LVARCHAR.

In both cases, the Unicode data is handled natively; it is not converted before manipulation. You might, however, need to cast Unicode to another character data type before using it in an application or some routines.

Unicode data in different columns may originate in different natural languages. With the Unicode DataBlade module, each column's data can remain in its original language if the database uses the generic Unicode locale that accompanies this DataBlade module. For more information on the generic Unicode locale, see "A Generic Unicode Locale" on page 2-5.

Important: *When you import Unicode data into your client program, you need to establish a way for the client's sort order to handle characters that may exist in the languages in the database but are not part of the character set specified for the client's locale.*

This DataBlade module also provides a utility, **gl_conv**, that you can use to convert binary data, such as spreadsheet files, to Unicode format. Once converted, this data can be stored in a UNIvarchar column. **gl_conv** also provides a way to load delimited Unicode data into your database from a file containing variable-width fields separated by a delimiter.

You can use the **gl_conv** utility to convert binary data, such as the data exported from a spreedsheet that is stored in a particular character set, to its Unicode equivalent format. Use the **gl_conv** utility to write the Unicode data to a file that contains variable length fields separated by a delimiter. The data can then be loaded into a database that contains UNIvarchar columns with the SQL LOAD command.

Handling Unicode Data

Informix DataBlade modules and other customer applications can create their own data types. The Unicode DataBlade module takes advantage of this facility to create a data type and supporting routines that can be used in SQL statements to store, retrieve, and manipulate data encoded in the Unicode character set.

This data type is called **UNIvarchar**.

Tip: *Note that it is the **data** that is in Unicode format. There are no Unicode databases or Unicode tables. Unicode data is stored in columns, so a column can have a UNIvarchar data type.*

What is in the Unicode DataBlade Module?

The Unicode DataBlade module extends the functionality of your Informix database server by providing the following features:

- The UNIvarchar data type
- Module-specific versions of database routines
- Module-specific casts
- The **gl_conv** utility

The UNIvarchar Data Type

The Unicode DataBlade module provides a new data type to store Unicode data. This data type is UNIvarchar.

Use of LVARCHAR

Dynamic Server with UD Option provides a built-in character data type, LVARCHAR, that can be used to store variable-length character data up to 2 kilobytes in length. LVARCHAR is used in casts and comparisons between two opaque types or between one opaque and one built-in type. It can be used to store multibyte strings.

LVARCHAR supports code-set collation of text data, as do CHAR and VARCHAR.

For details about the LVARCHAR data type, see the *Informix Guide to SQL: Syntax* and the *Informix Guide to SQL: Reference*.

This DataBlade module uses the LVARCHAR data type for Unicode data when it is being transmitted between client and server. When used with Unicode data, LVARCHAR stores the hexadecimal ASCII representation of Unicode. LVARCHAR data types must be cast to UNIvarchar or other character data types before they can be used in a database table. There are no LVARCHAR columns.

The LVARCHAR data type is also used for input and output casts for opaque data types. The LVARCHAR data type stores opaque data types in the string (external) format. Each opaque type has an input support function and cast, which convert it from LVARCHAR to a form that clients can manipulate. Each also has an output support function and cast, which convert it from its internal representation to LVARCHAR.

For examples of the hexadecimal representation of Unicode data, see the examples in Chapter 4, "Functions."

Customized Routines

Many of the supporting routines for data types have been redefined to work with UNIvarchar. These functions are documented in Chapter 4, "Functions."

Using the Unicode DataBlade Module

This section provides a high-level discussion of the use of the Unicode DataBlade module.

The Unicode Locale

In addition to the DB_LOCALE and CLIENT_LOCALE, the Unicode DataBlade module requires a Unicode locale for both the database and the client. For each client locale, the DataBlade module creates a compatible Unicode locale named *ll_tt*.unicode, where:

ll	Language. A two-character name that represents the natural language for a particular locale.
tt	Territory. A two-character name that represents the cultural conventions used in this locale. For example, the territory could indicate the Swiss version of French, Italian, or German.
unicode	The name of the codeset used by the locale.

This powerful feature provides flexibility and scope. To use it, however, you must be familiar with the idiosyncrasies of various languages. Each locale has its own characteristics, such as sort order, representation of dates, and legal identifiers. Informix has provided a generic locale with this DataBlade module, but you should set up your environment to handle data from different locales in a way that achieves your objectives.

See the release notes on the product media for details about the generic locale provided by Informix. For an in-depth discussion of locales and how to set up a non-default locale, see the *Informix Guide to GLS Functionality*.

Loading Data

The following sections describe how to load fixed-length and variable-length data into Unicode columns.

Loading Fixed-Length Data

You can use **Pload** to load and unload Unicode records if they are fixed length. Follow the instructions in the *Informix Guide to GLS Functionality*.

Loading Variable-Length Data

To load binary variable-length data into Unicode columns

1. Use **gl_conv** to convert the data to Unicode's external hexadecimal ASCII representation. The **gl_conv** utility is described in Chapter 6, "The gl_conv Utility."
2. Use the SQL LOAD statement to load the Unicode data into your database.

Using UNIvarchar Columns

A UNIvarchar column works just like a VARCHAR column except to retrieve data in Unicode format, you must write an ESQL/C program. DB-Access can only receive LVARCHAR data, which is a hexadecimal representation of Unicode data.

Otherwise, you can use SQL statements such as SELECT just as you would with any other data. See Chapter 5, "Module-Specific Syntax," for information about required casting in the SELECT statement.

Casting and Comparisons

Comparisons require casts to built-in data types. LVARCHAR is particularly useful for casting, as mentioned in "Use of LVARCHAR" on page 1-5.

To compare an opaque type to a built-in type, you must explicitly cast the opaque type to a form that the database server understands, such as LVARCHAR. The database server then invokes system-defined casts to convert the results to the desired built-in type.

To compare two opaque types, call the DataBlade module specific **Compare()** function in palce of the built-in **Compare()** function. The DataBlade module specific **Compare()** function takes as parameters two UNIvarchar data types. For more information on the **Compare()** function, see "COMPARE" on page 4-9.

See Chapter 3, "Casting and Unicode," to find out which casts are implicit and which are explicit, and to see examples of some explicit casts.

Chapter

Unicode Columns

2

In This Chapter

This chapter contains reference information for a new data type provided with the Informix Unicode DataBlade module. This data type, UNIvarchar, is used to store Unicode data in database tables.

This chapter also describes how the Unicode DataBlade module handles tables whose UNIvarchar columns contain data in different natural languages.

The UNIvarchar Data Type

The Unicode DataBlade module uses the UNIvarchar data type as the database representation for Unicode data.

Definition

UNIvarchar is an opaque data type created with the following modifiers to the CREATE OPAQUE TYPE statement.

CREATE OPAQUE TYPE Modifier	Value
internallength	variable
maxlen	1024 bytes (512 Unicode characters)
alignment	2

Tip: *Note that this data type can be up to 1024 bytes in length, but can hold only 512 Unicode characters since each Unicode character is 2 bytes in length.*

Usage

The UNIvarchar data type allows you to store native Unicode data in your database and manipulate it as you would any VARCHAR data. Assign the UNIvarchar data type to columns that contain data in Unicode format.

Using Built-in Server Functions

Some built-in server functions produce an inappropriate result when used with the UNIvarchar data type. Some of these functions have been replaced by module-specific user-defined functions, which are called in the place of the built-in functions. You should either avoid using the others or manipulate the results to use the external hexadecimal ASCII representation of the UNIvarchar data.

Functions that have been customized for this DataBlade module are described in Chapter 4, "Functions."

Locales

A UNIvarchar column has a Unicode code set associated with it. In other words, the data in different columns can originate in different locales—locales that are not the same as either the client locale or the server locale. The locale of a UNIvarchar column is based on the database's language and territory and on the Unicode data set.

The resulting locale is represented as *ll_tt*.unicode, where *ll* represents the code set language and *tt* represents the code set territory. The Unicode locale is described in "The Unicode Locale" on page 1-6.

The Unicode DataBlade module checks to see what the CTYPE is for the current locale. Then it dynamically creates the corresponding Unicode locale. For example, when the current locale is the default locale, en_us.8859-1, the Unicode DataBlade module sets its locale to en_us.unicode. All characters contained within the CTYPE category of this locale are available to the user.

If you need to use characters that are not contained within the CTYPE for you locale (Korean characters, for example), then you must set the current locale to the appropriate locale, such as ko_kr.KSC5601. The Unicode DataBlade module would then set the locale to ko_kr.unicode.

ESQL/C

To load a particular locale, use the GLS API function **gl_lc_load**. For example, the following statement loads the Unicode locale that corresponds to the default locale for your server, en_us.8859-1:

```
gl_lc_load(ul,"en_us.unicode", NULL)
```

♦

A Generic Unicode Locale

A generic Unicode locale, with the appropriate code map and conversion files, is distributed with this DataBlade module in a subdirectory of the installation root directory.

The locale file, **e005.lco**, contains LC_CTYPE information for the following character sets:

- ISO8859 characters sets from ISO8859-1 to ISO8859-9 inclusive
- Microsoft Windows characters sets CP1250 to CP1258 inclusive
- Asian character sets:
  - Chinese GB2312-1980
  - Japanese JIS\EUC 0201-1976, JIS\EUC 0208-1990, JIS\EUC 0212-1990
  - Korean KSC5601-1992

For more details and instructions on using this locale, see the release notes on the product media.

Important: *Informix strongly encourages you to use the module-provided locale. Doing so will greatly reduce the effort required to set up your environment.*

Special Considerations

Manipulating data from multiple character sets can require some extra planning. For example, if you want to use the characters in different character sets as object names and identifiers, you need to specify that in your CLIENT_LOCALE.

In addition, different character sets can have different sort orders, specified by the locale. You need to provide instructions for sorting characters that are not usually found in your character set.

See the *Informix Guide to GLS Functionality* for details.

Client Programs and Unicode Data

Client programs differ in the way they handle Unicode data.

ESQL/C

ESQL/C processes Unicode 16-bit data natively.

DB-Access

DB-Access cannot process 16-bit data. Instead of retrieving native Unicode data, it returns an LVARCHAR containing the hexadecimal ASCII representation of Unicode.

Additional Information

DB-Access, ESQL/C, and other client applications provide varying levels of support for localized data. Check the manual for your client application to determine whether it supports the use of localized character sets in names and allows you to sort data according to collation rules specified for the locale.

Chapter

Casting and Unicode

3

In This Chapter

This chapter contains reference information for the casts provided with the Informix Unicode DataBlade module. It contains reference pages for the explicit casts between the Unicode code set and the database code set. It also lists implicit and explicit casts that convert between internal and external representations of data.

The Importance of Casting

Timely casting is the key to successful use of the Unicode DataBlade module. Since UNIvarchar behaves much like VARCHAR, you can use built-in functions as long as you cast your data correctly. This chapter lists the implicit casts and describes the explicit casts that are available with the Unicode DataBlade module.

Implicit Casts

The following implicit casts convert between the external ASCII hexadecimal representation and the internal UNIvarchar representation.

From This External Representation	To This Internal Representation
LVARCHAR	UNIvarchar
SENDRECV	UNIvarchar
IMPEXP	UNIvarchar
IMPEXPBIN	UNIvarchar

Explicit Casts

The following explicit casts convert between the internal UNIvarchar representation and the external ASCII hexadecimal representation.

From This Internal Representation	To This External Representation
UNIvarchar	LVARCHAR
UNIvarchar	SENDRECV
UNIvarchar	IMPEXP
UNIvarchar	IMPEXPBIN

Cast Table

The casts in this section convert between the Unicode code set and the code set of the database. They must be explicit.

Use these casts to pass UNIvarchar data to module-defined or built-in functions that operate on data of built-in character data types.

For example, to use the built-in function MATCHES to determine whether a UNIvarchar value is `A` or `a`, you might use the following cast:

```
MATCHES (UNIvarchar::CHAR(30), "[aA]*")
```

The following table shows explicit casts to and from UNIvarchar.

This Cast	Returns
CHAR to UNIvarchar	UNIvarchar
NCHAR to UNIvarchar	UNIvarchar
VARCHAR to UNIvarchar	UNIvarchar
NVARCHAR to UNIvarchar	UNIvarchar
UNIvarchar to CHAR	CHAR

(1 of 2)

This Cast	Returns
UNIvarchar to NCHAR	NCHAR
UNIvarchar to NVARCHAR	NVARCHAR
UNIvarchar to VARCHAR	VARCHAR

(2 of 2)

Each of these casts is described on its own reference page, where you can find the support routines and examples for each.

Many casts done on Unicode data must be explicit. In some cases, such as quoted strings, you must cast the value to its character representation before casting it to UNIvarchar. This is necessary because a Unicode string is represented as a hexadecimal value (LVARCHAR), and you need to indicate that the quoted value is a character value not a hexadecimal value.

This double cast is illustrated in the second example provided for the casts described in this chapter.

CHAR to UNIvarchar

You must cast CHAR strings to UNIvarchar explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **UNIvarcharCastFromChar**.

Examples

The following examples illustrate the explicit cast from CHAR to UNIvarchar.

Setup

Create a table named **cars1** with four columns.

```
CREATE TABLE cars1
(
        id  INTEGER,
    model_c CHAR(10),
    model_U UNIvarchar,
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_c** columns. Because the value in the UNIvarchar column **model_U** is set to `NULL`, no cast is needed.

```
INSERT INTO cars1(id, model_c, model_U)
     VALUES
     (
        0,
        "Jaguar XJS",
        NULL
     );

INSERT INTO cars1(id, model_c, model_U)
     VALUES
     (
        1,
        "Fiat Ritmo",
        NULL
     );
```

Example 1

This example updates the data in the **cars1** table by setting the value of the **model_U** column, which is UNIvarchar, to be the same as the value in the **model_c** column, which is CHAR, for the row with an **id** of 0.

model_c must be cast to UNIvarchar before its data can be replicated in the **model_U** column.

```
UPDATE cars1
    SET model_U = model_c::UNIvarchar
    WHERE id = 0;
```

This statement queries the updated record.

```
SELECT * FROM cars1 WHERE id = 0;
```

The result is shown below. The UNIvarchar value is returned as a hexadecimal string.

```
id          0
model_c     Jaguar XJS
model_U     004A0061006700750061007200200058004A0053
error
```

Example 2

This example inserts data into the **model_U** column of row 5 directly, by specifying the value as a quoted string, cast to UNIvarchar.

```
INSERT INTO cars1(id, model_U)
    VALUES
    (
        5,"Jaguar XJS"::CHAR(10)::UNIvarchar
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars1 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_U** columns only. The **model_c** column for this row is empty.

```
id          5
model_c
model_U     004A0061006700750061007200200058004A0053
error
```

NCHAR to UNIvarchar

You must cast NCHAR strings to UNIvarchar explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **UNIvarcharCastFromNchar.**

Examples

The following examples illustrate the explicit cast from NCHAR to UNIvarchar.

Setup

Create a table named **cars2** with four columns.

```
CREATE TABLE cars2
(
        id  INTEGER,
    model_c NCHAR(10),
    model_U UNIvarchar,
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_c** columns. Because the value in the UNIvarchar column **model_U** is set to NULL, no cast is needed.

```
INSERT INTO cars2(id, model_c, model_U)
     VALUES
     (
        0,
        "Jaguar XJS",
        NULL
     );

INSERT INTO cars2(id, model_c, model_U)
     VALUES
     (
        1,
        "Fiat Ritmo",
        NULL
     );
```

Example 1

This example updates the data in the **cars2** table by setting the value of the **model_U** column, which is UNIvarchar, to be the same as the value in the **model_c** column, which is NCHAR, for the row with an **id** of 1.

```
UPDATE cars2
    SET model_U = model_c::UNIvarchar
    WHERE id = 1;
```

This statement queries the updated record.

```
SELECT * FROM cars2 WHERE id = 1;
```

The result is shown below. The UNIvarchar value is returned as a hexadecimal string.

```
id          1
model_c     Fiat Ritmo
model_U     00460069006100740020005200690074006D006F
error
```

Example 2

This example inserts data into the **model_U** column of row 5 directly, by specifying the value as a quoted string, cast to UNIvarchar.

```
INSERT INTO cars2(id, model_U)
    VALUES
    (
        5,"Jaguar XJS"::NCHAR(10)::UNIvarchar
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars2 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_U** columns only. The **model_c** column for this row is empty.

```
id          5
model_c
model_U     004A00610067007500610072002000580004A0053
error
```

NVARCHAR to UNIvarchar

You must cast NVARCHAR strings to UNIvarchar explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **UNIvarcharCastFromNvarchar.**

Examples

The following examples illustrate the explicit cast from NVARCHAR to UNIvarchar.

Setup

Create a table named **cars3** with four columns.

```
CREATE TABLE cars3
(
        id  INTEGER,
    model_c NVARCHAR(10),
    model_U UNIvarchar,
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_c** columns. Because the value in the UNIvarchar column **model_U** is set to `NULL`, no cast is needed.

```
INSERT INTO cars3(id, model_c, model_U)
     VALUES
     (
        0,
        "Jaguar XJS",
        NULL
     );

INSERT INTO cars3(id, model_c, model_U)
     VALUES
     (
        1,
        "Fiat Ritmo",
        NULL
     );
```

Example 1

This example updates the data in the **cars3** table by setting the value of the **model_U** column, which is UNIvarchar, to be the same as the value in the **model_c** column, which is NVARCHAR, for the row with an **id** of 0.

model_c must be cast to UNIvarchar before its data can be replicated in the **model_U** column. If the cast is omitted, an error will result.

```
UPDATE cars3
    SET model_U = model_c::UNIvarchar
    WHERE id = 0;
```

This statement queries the updated record.

```
SELECT * FROM cars3 WHERE id = 0;
```

The result is shown below. The UNIvarchar value is returned as a hexadecimal string.

```
id           0
model_c      Jaguar XJS
model_U      004A0061006700750061007200200058004A0053
error
```

Example 2

This example inserts data into the **model_U** column of row 5 directly, by specifying the value as a quoted string, cast to UNIvarchar.

```
INSERT INTO cars3(id, model_U)
    VALUES
    (
        5,"Jaguar XJS"::NVARCHAR(10)::UNIvarchar
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars3 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_U** columns only. The **model_c** column for this row is empty.

```
id          5
model_c
model_U     004A006100670075006100720020000058004A0053
error
```

VARCHAR to UNIvarchar

You must cast VARCHAR strings to UNIvarchar explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **UNIvarcharCastFromVarchar.**

Examples

The following examples illustrate the explicit cast from VARCHAR to UNIvarchar.

Setup

Create a table named **cars4** with four columns.

```
CREATE TABLE cars4
(
        id  INTEGER,
    model_c VARCHAR(10),
    model_U UNIvarchar,
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_c** columns. Because the value in the UNIvarchar column **model_U** is set to `NULL`, no cast is needed.

```
INSERT INTO cars4(id, model_c, model_U)
     VALUES
     (
        0,
        "Jaguar XJS",
        NULL
     );

INSERT INTO cars4(id, model_c, model_U)
     VALUES
     (
        1,
        "Fiat Ritmo",
        NULL
     );
```

Example 1

This example updates the data in the **cars4** table by setting the value of the **model_U** column, which is UNIvarchar, to be the same as the value in the **model_c** column, which is VARCHAR, for the row with an **id** of 1.

```
UPDATE cars4
    SET model_U = model_c::UNIvarchar
    WHERE id = 1;
```

This statement queries the updated record.

```
SELECT * FROM cars4 WHERE id = 1;
```

The result is shown below. Note that the UNIvarchar value is returned as a hexadecimal string.

```
id          1
model_c     Fiat Ritmo
model_U     0046006900610074002000520069007400 6D006F
error
```

Example 2

This example inserts data into the **model_U** column of row 5 directly, by specifying the value as a quoted string, cast to UNIvarchar.

```
INSERT INTO cars4(id, model_U)
    VALUES
    (
        5,"Jaguar XJS"::VARCHAR(10)::UNIvarchar
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars4 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_U** columns only. The **model_c** column for this row is empty.

```
id          5
model_c
model_U     004A0061006700750061007200200058004A0053
error
```

UNIvarchar to CHAR

You must cast UNIvarchar strings to CHAR explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **charCastFromUNIvarchar**.

Examples

The following examples illustrate the explicit cast from UNIvarchar to CHAR.

Setup

Create a table named **cars5** with four columns.

```
CREATE TABLE cars5
(
        id  INTEGER,
    model_U UNIvarchar,
    model_c CHAR(10),
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_U** columns. To be stored in the **model_U** column, the character data must be explicitly cast to UNIvarchar.

```
INSERT INTO cars5(id, model_U, model_c)
     VALUES
     (
        0,
        "Jaguar XJS"::UNIvarchar,
        NULL
     );

INSERT INTO cars5(id, model_U, model_c)
     VALUES
     (
        1,
        "Fiat Ritmo"::UNIvarchar,
        NULL
     );
```

Example 1

This example updates the data in the **cars5** table by setting the value of the **model_c** column, which is CHAR, to be the same as the value in the **model_U** column, which is UNIvarchar, for the row with an **id** of 0.

```
UPDATE cars5
    SET model_c = model_U::CHAR(10)
    WHERE id = 1;
```

This statement queries the updated record.

```
SELECT * FROM cars5 WHERE id = 1;
```

The result is shown below. Note that the UNIvarchar value is returned as a hexadecimal string.

```
id          1
model_U     004600690061007400200052006900740006D006F
model_c     Fiat Ritmo
error
```

Example 2

This example inserts data into the **model_c** column of row 5 directly, by specifying the value as a quoted string, cast to CHAR.

```
INSERT INTO cars5(id, model_c)
    VALUES
    (
        5,"Jaguar XJS"::UNIvarchar::CHAR(10)
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars5 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_c** columns only. The UNIvarchar column for this row is empty.

```
id          5
model_U
model_c     Jaguar XJS
error
```

UNIvarchar to NCHAR

You must cast UNIvarchar strings to NCHAR explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **ncharCastFromUNIvarchar**.

Examples

The following examples illustrate the explicit cast from UNIvarchar to NCHAR.

Setup

Create a table named **cars6** with four columns.

```
CREATE TABLE cars6
(
        id  INTEGER,
    model_U UNIvarchar,
    model_c NCHAR(10),
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_U** columns. To be stored in the **model_U** column, the character data must be explicitly cast to UNIvarchar.

```
INSERT INTO cars6(id, model_U, model_c)
     VALUES
     (
        0,
        "Jaguar XJS"::UNIvarchar,
        NULL
     );

INSERT INTO cars6(id, model_U, model_c)
     VALUES
     (
        1,
        "Fiat Ritmo"::UNIvarchar,
        NULL
     );
```

Example 1

This example updates the data in the **cars6** table by setting the value of the **model_c** column, which is NCHAR, to be the same as the value in the **model_U** column, which is UNIvarchar, for the row with an **id** of 0.

```
UPDATE cars6
    SET model_c = model_U::NCHAR(10)
    WHERE id = 0;
```

This statement queries the updated record.

```
SELECT * FROM cars6 WHERE id = 0;
```

The result is shown below. Note that the UNIvarchar value is returned as a hexadecimal string.

```
id          0
model_U     004A006100670075006100720020005800 4A0053
model_c     Jaguar XJS
error
```

Example 2

This example inserts data into the **model_c** column of row 5 directly, by specifying the value as a quoted string, cast to NCHAR.

```
INSERT INTO cars6(id, model_c)
    VALUES
    (
        5,"Jaguar XJS"::UNIvarchar::NCHAR(10)
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars6 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_c** columns only. The UNIvarchar column **model_U** for this row is empty.

```
id          5
model_U
model_c     Jaguar XJS
error
```

UNIvarchar to NVARCHAR

You must cast UNIvarchar strings to NVARCHAR explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **nvarcharCastFromUNIvarchar**.

Examples

The following examples illustrate the explicit cast from UNIvarchar to NVARCHAR.

Setup

Create a table named **cars7** with four columns.

```
CREATE TABLE cars7
(
        id  INTEGER,
    model_U UNIvarchar,
    model_c NVARCHAR(10),
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_U** columns. To be stored in the **model_U** column, the character data must be explicitly cast to UNIvarchar.

```
INSERT INTO cars7(id, model_U, model_c)
     VALUES
     (
        0,
        "Jaguar XJS"::UNIvarchar,
        NULL
     );

INSERT INTO cars7(id, model_U, model_c)
     VALUES
     (
        1,
        "Fiat Ritmo"::UNIvarchar,
        NULL
     );
```

Example 1

This example updates the data in the **cars7** table by setting the value of the **model_c** column, which is NVARCHAR, to be the same as the value in the **model_U** column, which is UNIvarchar, for the row with an **id** of 0.

```
UPDATE cars7
    SET model_c = model_U::NVARCHAR(10)
    WHERE id = 0;
```

This statement queries the updated record.

```
SELECT * FROM cars7 WHERE id = 0;
```

The result is shown below. The UNIvarchar value is returned as a hexadecimal string.

```
id          0
model_U     004A0061006700750061007200200058004A0053
model_c     Jaguar XJS
error
```

Example 2

This example inserts data into the **model_c** column of row 5 directly, by specifying the value as a quoted string.

```
INSERT INTO cars7(id, model_c)
    VALUES
    (
        5,"Jaguar XJS"::UNIvarchar::NVARCHAR(10)
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars7 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_c** columns only. The UNIvarchar column **model_U** for this row is empty.

```
id          5
model_U
model_c     Jaguar XJS
error
```

UNIvarchar to VARCHAR

You must cast UNIvarchar strings to VARCHAR explicitly. If you omit the explicit cast, an error will be returned.

The support routine for this cast is **varcharCastFromUNIvarchar**.

Examples

The following examples illustrate the explicit cast from UNIvarchar to VARCHAR.

Setup

Create a table named **cars8** with four columns.

```
CREATE TABLE cars8
(
        id  INTEGER,
    model_U UNIvarchar,
    model_c VARCHAR(10),
    error   LVARCHAR
);
```

Insert some data into the **id** and **model_U** columns. To be stored in the **model_U** column, the character data must be explicitly cast to UNIvarchar.

```
INSERT INTO cars8(id, model_U, model_c)
     VALUES
     (
        0,
        "Jaguar XJS"::UNIvarchar,
        NULL
     );

INSERT INTO cars8(id, model_U, model_c)
     VALUES
     (
        1,
        "Fiat Ritmo"::UNIvarchar,
        NULL
     );
```

Example 1

This example updates the data in the **cars8** table by setting the value of the **model_c** column, which is VARCHAR, to be the same as the value in the **model_U** column, which is UNIvarchar, for the row with an **id** of 0.

```
UPDATE cars8
    SET model_c = model_U::VARCHAR(10)
    WHERE id = 0;
```

This statement queries the updated record.

```
SELECT * FROM cars8 WHERE id = 0;
```

The result is shown below. Note that the UNIvarchar value is returned as a hexadecimal string.

```
id          0
model_U     004A00610067007500610072002000580004A0053
model_c     Jaguar XJS
error
```

Example 2

This example inserts data into the **model_c** column of row 5 directly, by specifying the value as a quoted string.

```
INSERT INTO cars8(id, model_c)
    VALUES
    (
        5,"Jaguar XJS"::UNIvarchar::VARCHAR(10)
    );
```

This statement queries the newly created record.

```
SELECT * FROM cars8 WHERE id = 5;
```

The result of this query shows that values have been inserted in the **id** and **model_c** columns only. The UNIvarchar column **model_U** for this row is empty.

```
id          5
model_U
model_c     Jaguar XJS
error
```

Using Casts with Date and Money Data Types

To convert a date or money data type to UNIvarchar, you must first cast it to a built-in character type, then cast it again to UNIvarchar.

For example, if **date_of_sale** is a column that contains DATE-type data, and you want to move the dates in that column into a UNIvarchar column, you must perform the following double cast:

```
date_of_sale::CHAR(30)::UNIvarchar
```

As with GLS, you must make any desired changes to the date format in your program.

Examples With[Without] a UNIvarchar Cast

The pairs of examples in this section illustrate the difference in the result when you cast to UNIvarchar and when you do not. The first example of the pair casts to UNIvarchar, the second one does not.

1. The following SELECT statement returns 09/11/97:

   ```
   SELECT
   (("9/10/97"::DATE)+1)::CHAR(20)::UNIvarchar::CHAR(20)
       AS DateToUvarchar
       FROM univartab
       WHERE itemno=1001;
   ```

   The following SELECT statement returns 09/11/97:

   ```
   SELECT (("9/10/97"::DATE)+1)::CHAR(20) AS DateToChar
       FROM univartab
       WHERE itemno=1001;
   ```

2. The following SELECT statement returns `09/10/97`:

   ```
   SELECT "9/10/97"::UNIvarchar::CHAR(20)::DATE
      AS UvarcharToDate
      FROM univartab
      WHERE itemno=1001;
   ```

   The following SELECT statement returns `09/10/97`:

   ```
   SELECT "9/10/97"::CHAR(20)::DATE AS CharToDate
      FROM univartab
      WHERE itemno=1001;
   ```

3. The following SELECT statement returns `$10.00`:

   ```
   SELECT "10"::MONEY::CHAR(20)::UNIvarchar::CHAR(20)
      AS MoneyToUvarchar
      FROM univartab
      WHERE itemno=1001;
   ```

   The following SELECT statement returns `$10.00`:

   ```
   SELECT "10"::MONEY::CHAR(20) AS MoneyToChar
      FROM univartab
      WHERE itemno=1001;
   ```

4. The following SELECT statement returns `$10.00`, right aligned in the column:

   ```
   SELECT "10"::UNIvarchar::CHAR(20)::MONEY
      AS UvarcharToMoney
      FROM univartab
      WHERE itemno=1001;
   ```

   The following SELECT statement returns `$10.00`, right aligned in the column:

   ```
   SELECT "10"::CHAR(20)::MONEY AS CharToMoney
      FROM univartab
      WHERE itemno=1001;
   ```

5. The following SELECT statement returns `10.0000000000000000`:

```
SELECT "10"::FLOAT::CHAR(20)::UNIvarchar::CHAR(20)
   AS FloatToUvarchar
   FROM univartab
   WHERE itemno=1001;
```

The following SELECT statement returns `10.0000000000000000`:

```
SELECT "10"::FLOAT::CHAR(20) AS FloatToChar
   FROM univartab
   WHERE itemno=1001;
```

6. The following SELECT statement returns `10.00000000000`:

```
SELECT "10"::UNIvarchar::CHAR(20)::FLOAT
   AS UvarcharToFloat
   FROM univartab
   WHERE itemno=1001;
```

The following SELECT statement returns `10.00000000000`:

```
SELECT "10"::CHAR(20)::FLOAT AS CharToFloat
   FROM univartab
   WHERE itemno=1001;
```

7. The following SELECT statement returns `10`:

```
SELECT "10.0"::INT::CHAR(20)::UNIvarchar::CHAR(20)
   AS IntToUvarchar
   FROM univartab
   WHERE itemno=1001;
```

The following SELECT statement returns `10`:

```
SELECT "10.0"::INT::CHAR(20) AS IntToChar
   FROM univartab
   WHERE itemno=1001;
```

8. The following SELECT statement returns 10, right aligned in the column:

```
SELECT "10.0"::UNIvarchar::CHAR(20)::INT
   AS UvarcharToInt
   FROM univartab
   WHERE itemno=1001;
```

The following SELECT statement returns 10, right aligned in the column:

```
SELECT "10.0"::CHAR(20)::INT AS CharToInt
   FROM univartab
   WHERE itemno=1001;
```

Chapter

Functions

4

In This Chapter

This chapter contains reference information for the functions provided with the Informix Unicode DataBlade module.

The following table summarizes the functions that are provided with this module.

Function	Description
CHAR_LENGTH	Counts the number of characters in a string, including trailing white spaces.
COMPARE	Compares two Unicode values.
CONCAT	Concatenates two Unicode expressions.
Equal	Compares two values and returns TRUE if they are equal.
GreaterThan	Compares two values and returns TRUE if the first is greater than the second.
GreaterThanOrEqual	Compares two values and returns TRUE if the first is greater than or equal to the second.
LENGTH	Calculates the length of a string in bytes, excluding trailing white spaces.
LessThan	Compares two values and returns TRUE if the first is less than the second.
LessThanOrEqual	Compares two values and returns TRUE if the first is less than or equal to the second.
NotEqual	Compares two values and returns TRUE if they are not equal.

(1 of 2)

Function	Description
OCTET_LENGTH	Calculates the length of a string in bytes, including trailing white spaces.
UNIvarcharExpB	Casts Unicode data to binary data.
UNIvarcharExpT	Calls UNIvarcharOutput.
UNIvarcharImpB	Casts binary data to Unicode for storage in a database table.
UNIvarcharImpT	Calls UNIvarcharInput.
UNIvarcharInput	Casts LVARCHAR data types to UNIvarchar.
UNIvarcharOutput	Casts UNIvarchar data types to LVARCHAR.
UNIvarcharReceive	Byte swap depends on server architecture.
UNIvarcharSend	Byte swap depends on client architecture.

(2 of 2)

CHAR_LENGTH

Determines the number of Unicode characters in a character column, string, or variable, including trailing white spaces.

Syntax

```
CHAR_LENGTH (string)
```

Arguments

Argument	Description	Data Type
string	The data whose length is being measured. This can be: ■ the name of a UNIvarchar column whose length is being measured. ■ a quoted string or any expression whose result is a string. This value must be cast explicitly to UNIvarchar. ■ the name of a variable (in ESQL/C and SPL functions only).	UNIvarchar

Usage

The **CHAR_LENGTH** function behaves just like the **LENGTH** function with one exception—it counts characters rather than bytes. **CHAR_LENGTH** returns the total number of *characters* in a UNIvarchar column, quoted string, or variable, including trailing white spaces as defined by the locale.

With Unicode data, the number of characters is different from the number of bytes. The **LENGTH** value of a double-byte string is two times the number of characters, whereas the **CHAR_LENGTH** value of the same string is the actual number of characters. Since each character occupies two bytes, a string that has 8 Unicode characters has a **CHAR_LENGTH** value of 8 and a LENGTH value of 16.

This function is called in place of the built-in **CHAR_LENGTH** function whenever the string whose character length is being measured has UNIvarchar as its data type.

See the *Informix Guide to GLS Functionality* for a full discussion of the use of the **CHAR_LENGTH** function with multibyte characters.

Tip: *It is not necessary to cast UNIvarchar characters to a built-in data type to execute **CHAR_LENGTH**. If you do, the function will not return the correct result. If the string is not already in UNIvarchar format, you must cast it to UNIvarchar. See the examples.*

Returns

This function returns an INTEGER value specifying the number of characters in the specified column, quoted string, or variable.

Type of String	Returns
Column	The number of characters in the column, including trailing white spaces
Quoted string	The number of characters in the string, including trailing white spaces
Variable	The number of characters contained in the variable, including any trailing white spaces, regardless of the defined length of the variable

Examples

Most of these examples are self-contained. Example 4 uses the **books** table described in Appendix A.

1. The following example returns 8:

   ```
   EXECUTE FUNCTION CHAR_LENGTH(“abcdefgh”::UNIvarchar);
   ```

   CHAR_LENGTH counts characters. There are eight characters in the quoted string.

2. The following example returns 11:

   ```
   EXECUTE FUNCTION CHAR_LENGTH("abcdefgh   "::UNIvarchar);
   ```

   CHAR_LENGTH counts trailing spaces, so the 3 spaces after the h in the quoted string are added to the 8 letter characters, for a total of 11 characters.

3. The following example returns 4:

   ```
   EXECUTE FUNCTION
   CHAR_LENGTH("0067007200E10020"::LVARCHAR::UNIvarchar);
   ```

   This example measures the character length of a hexadecimal string that is 8 bytes (4 double-byte characters) long. The **CHAR_LENGTH** function counts trailing spaces, so the space at the end, 0020, is counted.

   Because the original string is not in LVARCHAR format, it must first be cast to LVARCHAR, then to UNIvarchar.

4. The following example returns 15. It uses the **books** table, described in Appendix A.

   ```
   SELECT CHAR_LENGTH (title)
       FROM books
       WHERE bookid = "10005";
   ```

   This example returns the character length of the value in the UNIvarchar column, **title**, in the table, **books**, for the specified row. The value in that column is Le Petit Prince. The string has 13 letters and 2 spaces, or 15 characters in all.

See Also

"LENGTH" on page 4-29
"OCTET_LENGTH" on page 4-36

CHARACTER_LENGTH

This is another name for the **CHAR_LENGTH** function, which is described on page 4-5.

Syntax

```
CHARACTER_LENGTH (string)
```

COMPARE

Compares two Unicode substrings.

Syntax

```
COMPARE (string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Usage

COMPARE uses the GLS API function **ifx_gl_mbxcoll** to compare two Unicode strings. It uses the locale's collation order to determine if the value of one string it greater than or less than the value of the other.

This function is called in place of the built-in **COMPARE** function whenever the two strings being compared both have UNIvarchar as their data type.

Returns

COMPARE returns an INTEGER value that is:

- greater than 0 if *string1* is greater than *string2.*
- 0 if *string1* is equal to *string2.*
- less than 0 if *string1* is less than *string2.*

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns 0:

   ```
   SELECT title, COMPARE(title,"Me"::UNIvarchar)
       FROM books
       WHERE bookid = 10010;
   ```

   This example compares the quoted string Me with the value in the title column of the books table for the specified row. Since title is a UNIvarchar column, Me is cast to UNIvarchar. And because the title value in the specified row is also Me, the statement returns 0.

2. The following example returns a value less than 0:

   ```
   SELECT title, COMPARE(title,"Ms"::UNIvarchar)
       FROM books
       WHERE bookid = "10010";
   ```

   This example compares the quoted string Ms with the value in the **title** column. Because Me has a lower value than Ms, this statement returns a value that is less than 0.

3. The following example returns a value greater than 0:

   ```
   SELECT title, COMPARE(title,"He"::UNIvarchar)
       FROM books
       WHERE bookid = "10010";
   ```

   This example compares the quoted string He with the value in the **title** column. Because Me has a higher value than He, the resulting value is greater than 0.

4. The following example returns 0:

   ```
   SELECT title, COMPARE(title,"004D0065"::LVARCHAR)
       FROM books
       WHERE bookid = "10010";
   ```

   In this example, a hexadecimal string, which is the ASCII representation of a Unicode value, is explicitly cast to LVARCHAR, which can be compared to the UNIvarchar value in the **title** column because there is an implicit cast from LVARCHAR to UNIvarchar.

   004D0065 is the hexadecimal representation of the Unicode value, Me, so the two values are the same.

   To see how you must treat a quoted string that is not a hexadecimal value, see the examples under "CONCAT."

5. The following example returns 0.

This example is the same as the previous example, except that the explicit cast to LVARCHAR is omitted. The example still works because the quoted string defaults to LVARCHAR. The quoted string does not need to be cast to LVARCHAR because it is already an LVARCHAR value.

```
SELECT title, COMPARE(title,”004D0065”)
    FROM books
    WHERE bookid = “10010”;
```

This example compares the hexadecimal string for Me with the value in the **title** column. Both values are the same, so the result is 0.

See Also

The following functions call **COMPARE**:

“Equal” on page 4-15
“GreaterThan” on page 4-17
“GreaterThanOrEqual” on page 4-20
“LessThan” on page 4-25
“LessThanOrEqual” on page 4-27
“NotEqual” on page 4-34

CONCAT

Concatenates two UNIvarchar expressions.

Syntax

```
CONCAT(expr1, expr2)
```

Arguments

Argument	Description	Data Type
expr1	The first expression in the concatenated string	UNIvarchar
expr2	The second expression in the concatenated string	UNIvarchar

Usage

CONCAT is the operator function associated with the built-in concatenation operator, ||.

The **CONCAT** function allows you to concatenate two UNIvarchar expressions to produce a concatenated UNIvarchar value. The **CONCAT** function appends *expr2* to the end of *expr1*.

This function is called in place of the built-in **CONCAT** function whenever the two strings being compared both have UNIvarchar as their data type.

Returns

CONCAT returns a UNIvarchar string that is the concatenation of the two specified values.

Using CONCAT with Built-In Data Types

When you want to concatenate a UNIvarchar value with a value of a built-in data type, you need to perform a cast so that both values have the same data type. You can either cast the UNIvarchar value to the built-in data type or cast the built-in data type to UNIvarchar.

Because the data must be cast, there is no performance advantage in using this function to concatenate a UNIvarchar column with a column of some other data type.

Examples

These examples show how the **CONCAT** example in the *Informix Guide to SQL: Syntax* would look using the Unicode **CONCAT** function.

These examples use the **authors** table, described in Appendix A. In all cases, the result is a UNIvarchar value.

1. In this example, both expressions are UNIvarchar data, so you do not need to cast them within the **CONCAT** function. The result of the **CONCAT** function is cast to VARCHAR for readability. The example returns `EUFR631922France`.

```
SELECT CONCAT(codes, country)::VARCHAR(50)
    FROM authors
    WHERE authid = “A0121”;
```

2. In this example, only the **codes** value is a UNIvarchar value; the **name** value is a VARCHAR value and must be cast. The result of the **CONCAT** function is cast to VARCHAR for readability. The example returns `EUFR631922Antoine de St. Exupery`.

```
SELECT CONCAT(codes, name::UNIvarchar)::VARCHAR(50)
    FROM authors
    WHERE authid = “A0121”;
```

3. In this example, the literal value `A0121` must be cast to CHAR so that it is not considered a hexadecimal representation of the Unicode string.

   By default, a quoted string is treated as an LVARCHAR string and is assumed to contain four-digit hexadecimal representation of each Unicode character. (For example, an `A` would be represented as `0041`.) The value `A0121` is not a valid hexadecimal value and would therefore return an error. If a quoted character string were, by coincidence, a valid hexadecimal value, a result would be returned, but it would not be the expected result.

   In the example, the result of the **CONCAT** function is cast to VARCHAR for readability. The example returns `EUFR631922A0121`.

```
SELECT CONCAT(codes, “A0121”::CHAR(10)::UNIvarchar)::VARCHAR(50)
    FROM authors
    WHERE authid = “A0121”;
```

Equal

Compares two values to determine whether they are equal.

Syntax

```
Equal(string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Results

Equal returns:

- `t` (TRUE) if *string1* is equal to *string2*.
- `f` (FALSE) if *string1* is not equal to *string2*.

Usage

Use this function to compare two Unicode values when you want to know if *string1* is equal to *string2*.

The Unicode DataBlade module calls this function in place of the built-in **Equal** function whenever both strings being compared have UNIvarchar as their data type.

Equal is the operator function associated with the built-in operator, =.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns `f`:

   ```
   SELECT title, Equal(title,”He”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `He`, with the value in the **title** column of the **books** table for the specified row. Because the two values are not equal, the statement is false.

2. The following example returns `t`:

   ```
   SELECT title, Equal(title,”Me”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `Me`, with the value in the **title** column of the **books** table for the specified row. Because the two values are equal, the statement is true.

3. The following example uses the = operator and returns a **bookid** of `10010`. Note that the quoted string is the four-digit hexadecimal representation of `Me`.

   ```
   SELECT bookid
       FROM books
       WHERE title = “004D0065”;
   ```

See Also

“COMPARE” on page 4-9
“NotEqual” on page 4-34

GreaterThan

Compares two values to determine whether the first is greater than the second.

Syntax

```
GreaterThan(string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Results

GreaterThan returns:

- `t` (TRUE) if *string1* is greater than *string2.*
- `f` (FALSE) if *string1* is less than or equal to *string2.*

Usage

Use this function to compare two Unicode values when you want to know if *string1* is greater than *string2*.

This function is called in place of the built-in **GreaterThan** function whenever the two strings being compared both have UNIvarchar as their data type.

GreaterThan is the operator function associated with the built-in operator, >.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns `f`:

   ```
   SELECT title, GreaterThan(title,”Ms”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string `Ms` with the value in the **title** column of the **books** table for the specified row. Because the first value (`Me`) is less than the second (`Ms`), the statement is false.

2. The following example returns `f`:

   ```
   SELECT title, GreaterThan(title,”Me”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string `Me` with the value in the **title** column of the **books** table for the specified row. Because the two values are equal, this statement is false.

3. The following example returns `t`:

   ```
   SELECT title, GreaterThan(title,”He”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string `He` with the value in the **title** column of the **books** table for the specified row. Because the first value (`Me`) is greater than the second (`He`), the statement is true.

4. The following example returns the **bookid**, `10012`. Note that **title**, which is a UNIvarchar column, is represented in four-digit hexadecimal format.

   ```
   SELECT bookid
       FROM books
       WHERE title > “004D0065”;
   ```

See Also

“COMPARE” on page 4-9
“GreaterThanOrEqual” on page 4-20
“LessThan” on page 4-25

GreaterThanOrEqual

Compares two values to determine whether the first is greater than or equal to the second.

Syntax

```
GreaterThanOrEqual(string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Results

GreaterThanOrEqual returns:

- `t` (TRUE) if *string1* is greater than or equal to *string2*.
- `f` (FALSE) if *string1* is less than *string2*.

Usage

Use this function to compare two Unicode values when you want to know if *string1* is greater than or equal to *string2*.

This function is called in place of the built-in **GreaterThanOrEqual** function whenever the two strings being compared both have UNIvarchar as their data type.

GreaterThanOrEqual is the operator function associated with the built-in operator, >=.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns f:

   ```
   SELECT title, GreaterThanOrEqual(title,”Ms”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, Ms, with the value in the **title** column of the **books** table for the specified row. Because the first value (Me) is less than the second (Ms), the statement is false.

2. The following example returns t:

   ```
   SELECT title, GreaterThanOrEqual(title,”Me”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, Me, with the value in the **title** column of the **books** table for the specified row. Because the two values are equal, this function is true.

3. The following example returns t:

   ```
   SELECT title, GreaterThanOrEqual(title,”He”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, He, with the value in the **title** column of the **books** table for the specified row. Because the first value (Me) is greater than the second (He), the statement is true.

4. The following example returns two **bookid** values: 10010 and 10012:

   ```
   SELECT bookid
   FROM books
   WHERE title >= “004D0065”;
   ```

   The hexadecimal value 004E0065 represents the **title** value Me. Two titles in the **books** table meet the >= 004E0065 criteria: Me and She.

See Also

"COMPARE" on page 4-9
"GreaterThan" on page 4-17
"LessThanOrEqual" on page 4-27

Hash

Performs the hash operation for Unicode data.

Syntax

```
Hash(string)
```

Arguments

Argument	Description	Data Type
string	The data whose substring is being specified. This value can be: ■ the name of the column whose data length is being measured. ■ a quoted literal string. ■ the name of a variable (in ESQL/C and SPL functions only).	UNIvarchar

Usage

The server uses the Hash function to cache function return values and execute distinct aggregate queries.

Bit-hashable types have the property:

If A = B then hash(A) = hash(B)

Since UNIvarchar is based on the VARCHAR data type, it must follow the SQL rule that requires that trailing blank spaces be ignored in equality comparisons.

Thus two UNIvarchar values with different numbers for trailing blank spaces have different bit representations but should be considered equal.

Returns

This function returns the hash value for the specified string or column. The returned value is an integer.

LessThan

Compares two values to determine whether the first is less than the second.

Syntax

```
   LessThan(string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Results

LessThan returns:

- `t` (TRUE) if *string1* is less than *string2*
- `f` (FALSE) if *string1* is greater than or equal to *string2*

Usage

Use this function to compare two Unicode values when you want to know if *string1* is less than *string2*.

This function is called in place of the built-in **LessThan** function whenever the two strings being compared both have UNIvarchar as their data type.

LessThan is the operator function associated with the built-in operator, <.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns `t`:

   ```
   SELECT title, LessThan(title,”Ms”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `Ms`, with the value in the **title** column of the **books** table for the specified row. Because the first value (`Me`) is less than the second (`Ms`), the statement is true.

2. The following example returns `f`:

   ```
   SELECT title, LessThan(title,”Me”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `Me`, with the value in the **title** column of the **books** table for the specified row. Because the two values are equal, this function is false.

3. The following example returns `f`:

   ```
   SELECT title, LessThan(title,”He”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `He`, with the value in the **title** column of the **books** table for the specified row. Because the first value (`Me`) is greater than the second (`He`), the statement is false.

4. The following example returns two **bookid** values: `10005` and `10011`:

   ```
   SELECT bookid
       FROM books
       WHERE title <“004D0065”;
   ```

See Also

“COMPARE” on page 4-9
“LessThanOrEqual” on page 4-27
“GreaterThan” on page 4-17

LessThanOrEqual

Compares two values to determine whether the first is less than or equal to the second.

Syntax

```
LessThanOrEqual(string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Results

LessThanOrEqual returns:

- `t` (TRUE) if *string1* is less than or equal to *string2*.
- `f` (FALSE) if *string1* is greater than *string2*.

Usage

Use this function to compare two Unicode values when you want to know if *string1* is less than or equal to *string2*.

This function is called in place of the built-in **LessThanOrEqual** function whenever the two strings being compared both have UNIvarchar as their data type.

LessThanOrEqual is the operator function associated with the built-in operator, <=.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns `t`:

   ```
   SELECT title, LessThanOrEqual(title,"Ms"::UNIvarchar)
       FROM books
       WHERE bookid = "10010";
   ```

   This example compares the quoted string, `Ms`, with the value in the **title** column of the **books** table for the specified row. Because the first value (`Me`) is less than the second (`Ms`), the statement is true.

2. The following example returns `t`:

   ```
   SELECT title, LessThanOrEqual(title,"Me"::UNIvarchar)
       FROM books
       WHERE bookid = "10010";
   ```

   This example compares the quoted string, `Me`, with the value in the **title** column of the **books** table for the specified row. Because the two values are equal, this function is true.

3. The following example returns `f`:

   ```
   SELECT title, LessThanOrEqual(title,"He"::UNIvarchar)
       FROM books
       WHERE bookid = "10010";
   ```

   This example compares the quoted string, `He`, with the value in the **title** column of the **books** table for the specified row. Because the first value (`Me`) is greater than the second (`He`), the statement is false.

4. The following example returns three **bookids:** `10005`, `10010` and `10011`:

   ```
       SELECT bookid
           FROM books
           WHERE title <="004D0065";
   ```

See Also

"COMPARE" on page 4-9
"LessThan" on page 4-25
"GreaterThanOrEqual" on page 4-20

LENGTH

Calculates the length, in bytes, of the data in a column, string, or variable, excluding trailing white spaces.

Syntax

```
LENGTH(data)
```

Arguments

Argument	Description	Data Type
data	The data whose length is being measured. This value can be: ■ the name of the column whose data length is being measured. ■ a quoted literal string or any expression whose result is a literal string ■ the name of a variable (in ESQL/C and SPL functions only).	UNIvarchar

Usage

The **LENGTH** function returns the number of *bytes* of data contained in a character column, a quoted string, or a variable, excluding trailing white spaces as defined by the locale.

When used with Unicode data, the result—the number of bytes— is different from the number of characters. Since each character occupies two bytes, the **LENGTH** of a double-byte string is two times the number of characters. That is, a Unicode string that has 8 Unicode characters and does not end with a space will have a **LENGTH** of 16.

To include white spaces in the length calculation, use **OCTET_LENGTH**.

See the *Informix Guide to GLS Functionality* for a discussion of the use of the **LENGTH** function with multibyte characters.

Tip: *It is not necessary to cast UNIvarchar characters to a built-in data type to execute* ***LENGTH****. If you do, the function will not return the correct result. If the string is not already in UNIvarchar format, you must cast it to UNIvarchar. See the examples.*

Returns

This function returns the following values:

Type of string	Returns
Column	The length of the character data in the column in bytes, excluding trailing spaces, regardless of the defined length of the column.
Quoted string	The number of bytes included within the quotation marks, minus any bytes used by trailing spaces as defined in the locale.
Variable	The number of bytes in the data contained in the variable, minus trailing spaces, regardless of the defined length of the variable.

Examples

Most of these examples do not use tables. Example 4 uses the **books** table described in Appendix A.

1. The following example returns `16`:

   ```
   EXECUTE FUNCTION LENGTH(“abcdefgh”::UNIvarchar);
   ```

   In Unicode, all characters, including alphabetic characters, are double-byte. That is, each letter in the literal string `abcdefg` is 2 bytes long. Since the **LENGTH** function counts bytes, not characters, the 8-character string has a length of 16.

2. The following example returns `16`:

   ```
   EXECUTE FUNCTION LENGTH(“abcdefgh   “::UNIvarchar);
   ```

   The **LENGTH** function ignores trailing spaces, so the three spaces after the `h` are not counted. Therefore, “`abcdefg`” returns the same length as “`abcdefg   `”.

3. The following example returns 6:

```
EXECUTE FUNCTION
LENGTH("0067007200E10020"::LVARCHAR::UNIvarchar);
```

   This example measures the length of a hexadecimal string that is 8 bytes long. (A byte is represented by a character pair, for example, E1.) A double-byte Unicode character is represented by two character pairs (for example, 00E1.) A space in Unicode is represented by the value 0020, which is the last double-byte character in the string. Since the **LENGTH** function deletes trailing spaces, these two bytes are not counted. Therefore, the length of the string is 8 bytes - 2 bytes = 6 bytes.

   Because the string is not originally in LVARCHAR format, it must first be cast to LVARCHAR, then to UNIvarchar.

4. The following example returns 30. It refers to the **books** table described in Appendix A.

```
SELECT LENGTH (title)
    FROM books
    WHERE bookid = "10005";
```

   This example returns the number of bytes in the UNIvarchar column, **title**, in the table, **books**. The value of **title** in the specified row is Le Petit Prince. Since the spaces in the string are not at the end of the string, they are counted. The string has 13 letters and two spaces, or 15 double-byte characters in all, for a total length of 30 bytes.

See Also

"OCTET_LENGTH" on page 4-36
"CHAR_LENGTH" on page 4-5

MATCHES

Compares two character strings, optionally using wild cards.

Syntax

```
WHERE string1 MATCHES string2
```

Arguments

Argument	Description	Data Type
string1	The string that is being compared to *string2*	Any character data type, including UNIvarchar
string1	The string to which *string1* is being compared	Any character data type, including UNIvarchar

Usage

The Unicode DataBlade module uses the built-in **MATCHES** operator. There is currently no module-specific operator or function to be called in its place.

As a result, you must cast UNIvarchar values to a built-in character type before they are compared using a **MATCHES** statement.

Example

This example demonstrates the use of casts when using **MATCHES**. The returned values follow the example.

This example uses the table **univartab** described in Appendix A.

```
SELECT itemno, car::char(30), plane::char(30)
    FROM univartab
    WHERE car::char(30) MATCHES “[aA][bB][Cc]”;
```

This statement returns the following data:

```
itemno  (expression) (expression)

  2001  abc          Tax
  2001  ABC          T?x
```

The **car** and **plane** columns, which are UNIvarchar columns, have been cast to a built-in character data type, so that the built-in **MATCHES** function can be used in the WHERE clause.

See the documentation for your database server and *Informix Guide to SQL: Syntax* to learn more about how **MATCHES** works.

NotEqual

Compares two values to determine whether they are not equal.

Syntax

```
NotEqual(string1, string2)
```

Arguments

Argument	Description	Data Type
string1	The first of the two strings being compared	UNIvarchar
string2	The second of the two strings being compared	UNIvarchar

Results

NotEqual returns:

- `f` (FALSE) if *string1* is equal to *string2.*
- `t` (TRUE) if *string1* is not equal to*string2.*

Usage

Use this function to compare two Unicode values when you want to know if string1 is not equal to string2.

This function is called in place of the built-in **NotEqual** function whenever the two strings being compared both have UNIvarchar as their data type.

NotEqual is the operator function associated with the built-in operator, !=.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns `t`:

   ```
   SELECT title, NotEqual(title,”Ms”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `Ms`, with the value in the **title** column of the **books** table for the specified row. Because the two values are not equal, the statement is true.

2. The following example returns `f`:

   ```
   SELECT title, NotEqual(title,”Me”::UNIvarchar)
       FROM books
       WHERE bookid = “10010”;
   ```

   This example compares the quoted string, `Me`, with the value in the **title** column of the **books** table for the specified row. Because the two values are equal, the statement is false.

3. The following example returns all **bookid** values except `10010`, namely, `10005`, `10011`, and `10012`:

   ```
   SELECT bookid
       FROM books
       WHERE title !=”004D0065”;
   ```

See Also

“COMPARE” on page 4-9
“Equal” on page 4-15

OCTET_LENGTH

Determines the length, in bytes, of the data in a column, string, or variable, including trailing white spaces.

Syntax

```
OCTET_LENGTH(data)
```

Arguments

Argument	Description	Data Type
data	The data whose length is being measured. This can be: ■ the name of a UNIvarchar column whose length is being measured. ■ a quoted string or any expression whose result is a string ■ the name of a variable (in ESQL/C and SPL functions only).	UNIvarchar

Usage

The **OCTET_LENGTH** function behaves just like the **LENGTH** function with one exception: trailing white spaces—as defined in the locale—are included in the count. It returns the total number of bytes contained in a character column, a quoted string, or a variable.

When the data is Unicode data, the result—the number of bytes— is different from the number of characters. Since each character occupies 2 bytes, the **OCTET_LENGTH** of a double-byte string is two times the number of characters. That is, a Unicode string that has 8 Unicode characters has an **OCTET_LENGTH** of 16.

To exclude trailing white spaces from the length calculation, use **LENGTH**.

See the *Informix Guide to GLS Functionality* for a discussion of the use of the **LENGTH** function with multibyte characters.

Tip: *It is not necessary to cast* UNIvarchar *characters to a built-in data type to execute* ***OCTET_LENGTH****. If you do, the function will not return the correct result. If the string is not already in* UNIvarchar *format, you must cast it to* UNIvarchar*. See the examples.*

Returns

This function returns the number of bytes in the specified column, quoted string, or variable.

Type of string	Returns
Column	The defined length of the column in bytes, regardless of the number of bytes actually stored in the column.
Quoted string	The number of bytes inside the quotation marks, including trailing white spaces
Variable	The length of the data contained in the variable, including any trailing white spaces, regardless of the defined length of the variable.

Examples

Most of the following examples are self-contained. Example 4 uses the **books** table described in Appendix A.

1. The following example returns 16:

   ```
   EXECUTE FUNCTION OCTET_LENGTH(“abcdefgh”::UNIvarchar);
   ```

   OCTET_LENGTH counts bytes. There are eight double-byte characters, and therefore 16 bytes, in the quoted string.

2. The following example returns 22:

   ```
   EXECUTE FUNCTION OCTET_LENGTH("abcdefgh   "::UNIvarchar);
   ```

   OCTET_LENGTH counts trailing spaces, so the three spaces (6 bytes) after the h are added to the 16 bytes occupied by the letter characters, for a total of 22 bytes.

3. The following example returns 8:

   ```
   EXECUTE FUNCTION
   OCTET_LENGTH("0067007200E10020"::LVARCHAR::UNIvarchar);
   ```

   This example measures the length of a hexadecimal string that is 8 bytes (four double-byte characters) long. The **OCTET_LENGTH** function counts trailing spaces, so the space at the end, 0020, is counted.

   Because the original string is not in LVARCHAR format, it must first be cast to LVARCHAR, then to UNIvarchar.

4. The following example returns 30. (See the example under **LENGTH** for the table structure and the values in the specified row.)

   ```
   SELECT OCTET_LENGTH (title)
       FROM books
       WHERE bookid = "10005";
   ```

   This example returns the number of bytes in a UNIvarchar column, **title**, in the table, **books**. The value in that column is Le Petit Prince. The string has 13 letters and two spaces, or 15 double-byte characters in all, for a total octal length of 30 bytes.

See Also

"LENGTH" on page 4-29
"CHAR_LENGTH" on page 4-5

UNICharSubstring

Performs the substring ([...]) operation for Unicode data, by specifying character position.

Syntax

```
UNICharSubstring(string, first, last)
```

Arguments

Argument	Description	Data Type
string	The data whose substring is being specified. This value can be: ■ The name of the column whose data length is being measured. ■ A quoted literal string. ■ The name of a variable (in ESQL/C and SPL functions only).	UNIvarchar
first	The position of the first character of the specified substring. The first character in the string is in Position 1. If characters 3 through 8 in a 10-byte string are specified, this value is 3.	INTEGER
last	The position of the last character of the specified substring. The first byte in the string is in Position 1. If characters 3 through 8 in a 10-byte string are specified, this value is 8.	INTEGER

Returns

UNICharSubstring returns UNIvarchar.

Usage

Use **UNICharSubstring** to specify a substring of a UNIvarchar string by character position. Using the syntax shown above, specify the position of the first and last characters in the desired substring. The substring will be returned in UNIvarchar format.

The first character in the string is in Position 1.

This differs from the behavior of the built-in substring operator, which specifies byte number rather than character number. If you prefer to specify byte numbers with UNIvarchar data, use **UNIOctetSubstring**.

Examples

Examples 2 and 3 use the **customers** table, which is described in Appendix A.

1. The following example returns `0063006400650066`.

   ```
   EXECUTE FUNCTION
   UNICharSubstring("abcdefgh"::UNIvarchar,3,6);
   ```

   This example selects the third, fourth, fifth, and sixth characters in the string, `abcdefgh`. These characters are `cdef`, whose hexadecimal representation is `0063006400650066`.

2. The following example returns `00390034003100310031005300460043004100 30`, which is the hexadecimal representation of `94111SFCA0`.

   ```
   SELECT UNICharSubstring(codes, 1,10) FROM customers
       WHERE custid = "00123";
   ```

   This example returns the first 10 characters of the **codes** column in the **customers** table.

3. The following example returns `00390034003100310031`, which is the hexadecimal representation of `94111`.

   ```
   SELECT UNICharSubstring(codes, 1,5) FROM customers
       WHERE custid = "00123";
   ```

   This statement returns only the zip code, which occupies the first five characters of value in the **codes** column in the **customers** table.

See Also

“UNIOctetSubstring” on page 4-42

UNIOctetSubstring

Performs the substring ([...]) operation for Unicode data, by specifying byte position.

Syntax

```
UNIOctetSubstring(string, first, last)
```

Arguments

Argument	Description	Data Type
string	The data whose substring is being specified. This value can be: ■ the name of the column whose data length is being measured. ■ a quoted literal string. ■ the name of a variable (in ESQL/C and SPL functions only).	UNIvarchar
first	The position of the first byte of the specified substring. The first byte in the string is in Position 1. If bytes 3 through 8 in a 10-byte string are specified, this value is 3.	INTEGER
last	The position of the last byte of the specified substring. The first byte in the string is in Position 1. If bytes 3 through 8 in a 10-byte string are specified, this value is 8.	INTEGER

Returns

UNIOctetSubstring returns UNIvarchar.

Usage

Use **UNIOctetSubstring** to specify a substring of a UNIvarchar string by byte position. Using the syntax shown, specify the position of the first and last bytes in the desired substring. The substring will be returned in UNIvarchar format.

The first byte in the string is in Position 1.

This duplicates the behavior of the built-in substring operator, [...], which is described in the *Informix Guide to GLS Functionality*. It specifies byte number rather than character number. If you prefer to specify character numbers with UNIvarchar data, use **UNICharSubstring**.

Examples

The first example is self-contained. The second example uses the **customers** table described in Appendix A.

1. The following example returns `00620063`:

   ```
   EXECUTE FUNCTION
   UNIOctetSubstring("abcdefgh"::UNIvarchar,3,6);
   ```

   When the string `abcdefgh` is cast to UNIvarchar, each of the letters becomes a double-byte character. The letter `a`, whose hexadecimal value is `61`, becomes `0061` and occupies 2 bytes. The hexadecimal representation of the entire string is: `00610062006300640065006600670068`:

   The example specifies bytes 3 through 6. The third and fourth bytes are `0062`; the fifth and sixth bytes are `0063`, so the result is `00620063`.

2. The following example returns `0039003400310031 0031`, which is the hexadecimal representation of `94111`:

   ```
   SELECT UNIOctetSubstring(codes, 1,10) FROM customers
   WHERE custid = "00123";
   ```

   This example returns a five-digit zip code, which occupies the first five characters (10 bytes) of the **codes** column in the **customers** table.

See Also

"UNICharSubstring" on page 4-39

UNITrim

Removes trailing spaces from UNIvarchar data.

Syntax

```
UNITrim(expr)
```

Arguments

Argument	Description	Data Type
expr	The expression whose trailing spaces are to be removed	UNIvarchar

Usage

This function is a simplified version of the built-in SQL **TRIM** function.

The built-in **TRIM** function strips both leading and trailing pad characters. **UNITrim** removes only trailing spaces. To remove leading pad characters, or to specify a different pad character to be removed, cast the UNIvarchar data to a built-in data type such as CHAR or VARCHAR (not LVARCHAR) and perform a **TRIM** operation on the converted data. Be aware, however, that this conversion has a performance cost.

Returns

UNITrim returns a UNIvarchar string that is identical to the original string except that any trailing pad characters are removed.

Examples

These examples use the **books** table described in Appendix A.

1. The following example returns the hexadecimal value for the trimmed title 0043006F0073006D006F0073:

   ```
   SELECT UNITrim(title)::UNIvarchar
       FROM books
       WHERE bookid = “10011”;
   ```

   Compare this to a simple SELECT without the **UNITrim** function, which would return 0043006F0073006D006F0073002000200020002000200020:

   ```
   SELECT title::UNIvarchar
       FROM books
       WHERE bookid = “10011”;
   ```

2. The following example returns the number of characters in the trimmed title, which is 6:

   ```
   SELECT CHAR_LENGTH(UNITrim(title))
       FROM books
       WHERE bookid = “10011”;
   ```

   Compare this to a similar length operation on the untrimmed value, which returns 12:

   ```
   SELECT CHAR_LENGTH(title)
       FROM books
       WHERE bookid = “10011”;
   ```

Chapter

Module-Specific Syntax

5

In This Chapter

This chapter discusses the SQL syntax that is specific to the Informix Unicode DataBlade module. It contains reference information about additions and changes to existing SQL statements.

CREATE TABLE	This section explains how to use CREATE TABLE with this DataBlade module and what restrictions are placed upon its use.	page 5-4
SELECT	This section discusses when and how to use casts in the WHERE clause of the SELECT statement when querying Unicode data. It also lists customized and unsupported functions and operators.	page 5-6

For the complete syntax of these statements, see the *Informix Guide to SQL: Syntax*.

General Information

The UNIvarchar data type is designed to behave like the VARCHAR data type. Therefore, you can use UNIvarchar just like a VARCHAR in SQL statements.

CREATE TABLE

When you create a table to handle Unicode data, you must specify UNIvarchar as the data type in the Column Definition Clause for all columns that will hold Unicode data.

Syntax

Use this syntax when creating columns to hold Unicode data:

```
   CREATE TABLE table_name (column_name UNIvarchar...)...
```

This and other module-specific syntax rules are shown in the following table.

Clause	Use with Unicode DataBlade Module	Cross-References
Column Definition Clause / Data Type	Use UNIvarchar as the data type in the column definition clause for all columns that contain Unicode data. Do not use the UNIvarchar data type for a column that is a key.	"The UNIvarchar Data Type" on page 2-3 The *Administrator's Guide* for your database server
Table type	Any.	
Storage Option / IN clause	You cannot store UNIvarchar data in an extspace. It must be the default dbspace or a separate dbspace or an sbspace.	CREATE TABLE reference pages in the *Informix Guide to SQL: Syntax*
Table-Level Constraints	Same as any other table.	CREATE TABLE reference pages in the *Informix Guide to SQL: Syntax*

Usage

The column definition shown in this section creates a UNIvarchar column in a database table. Once created, a UNIvarchar column behaves like a VARCHAR column.

Tip: *Do not use a UNIvarchar column as an index key. The maximum length of a key value in an index is 390 bytes. The value of a UNIvarchar column is 512 bytes. The attempt to set the index key would fail and an error would be returned. For details, see the Administrator's Guide for your Informix database server.*

SELECT

Use the SELECT statement to query data in UNIvarchar format.

There are no changes in the way you write SELECT statements for UNIvarchar data. Treat UNIvarchar just like a VARCHAR in SQL statements.

When your SELECT statement compares UNIvarchar and regular VARCHAR data, the Unicode DataBlade module performs an explicit cast of the VARCHAR data to UNIvarchar before doing the comparison. This action allows selects on UNIvarchar columns to appear no different from VARCHAR data. The cast is transparent to the user.

Tip: *Because of the nature of UNIvarchar data, this cast is actually a code set conversion.*

The WHERE Clause

When your SELECT statement compares a UNIvarchar column with a column of a different data type, a cast must be performed so that both compared items have the same data type. (In the Unicode DataBlade module, this cast is actually a code set conversion.) The Unicode DataBlade module has casts from UNIvarchar to built-in character data types, so this cast is transparent to the user.

For example, the following statement performs an explicit cast from VARCHAR to UNIvarchar and returns the **bookid** 10010:

```
SELECT bookid,title
    FROM books
    WHERE title = “Me”::UNIvarchar;
```

In the following example, the quoted string, 004800610069006b0075, the hexadecimal ASCII representation of the word `haiku`, is explicitly cast to LVARCHAR in the INSERT statement. Because **title** is a UNIvarchar column, an explicit cast from UNIvarchar to VARCHAR, the data type of the value `Haiku`, must take place. This example returns the **bookid** `10013`:

```
INSERT INTO books
    VALUES (“10013”,
    “004800610069006b0075”::LVARCHAR,
    “Saitoh”,
    “A book of Japanese Haiku poems”
);

SELECT bookid, title
    FROM books
    WHERE title::VARCHAR(5) = “Haiku”;
```

WHERE clauses that compare UNIvarchar columns with non-UNIvarchar columns are subject to an implicit or explicit cast whereby one column type is converted to the other. For example, comparing a UNIvarchar with a VARCHAR may result in a cast (a code set conversion in this case) of the VARCHAR column to a UNIvarchar column. This is transparent to the user and is normal Datablade module execution.

Unsupported Operators

Because of the nature of the UNIvarchar data type, the following operators should be used with caution. When you use one of these operators with UNIvarchar data, you must cast the UNIvarchar data to a built-in character data type (CHAR, VARCHAR, NCHAR, NVARCHAR) in order to return correct results.

- MATCHES
- DATE
- LIKE

Custom Operators

Some operators and functions return results that are more appropriate for single-byte data than for double-byte Unicode data. In these cases, the Unicode DataBlade module provides custom operators and functions that return more appropriate results. These operators and functions are listed here and described in Chapter 4, "Functions."

The following operators have functionality specific to the Unicode DataBlade module. Their behavior differs from that of the corresponding server function.

Function	Related Operator	UNIvarchar function	Cross-Reference
CHAR_LENGTH		CHAR_LENGTH	"CHAR_LENGTH" on page 4-5
CONCAT	| |	CONCAT()	"CONCAT" on page 4-12
DATE		Must be cast	"Using Casts with Date and Money Data Types" on page 3-27
LENGTH		LENGTH (UNIvarchar)	"LENGTH" on page 4-29
OCTET_LENGTH		OCTET_LENGTH (UNIvarchar)	"OCTET_LENGTH" on page 4-36
SUBSTRING	[ ]	UNIOctetSubstring or UNICharSubstring	"UNIOctetSubstring" on page 4-42, and "UNICharSubstring" on page 4-39
TRIM		UNITrim	"UNITrim" on page 4-45

All other SQL operations, such as IN, BETWEEN, and relational operators work normally on UNIcode data.

See the *Informix Guide to SQL: Syntax* for details on standard relational operators.

Chapter

The gl_conv Utility

6

In This Chapter

This chapter describes the **gl_conv** utility and provides instructions for using it.

gl_conv

gl_conv is a utility that is shipped with the Informix Unicode DataBlade module. It enables you to convert binary data to Unicode, so that you can store the data in your database. If the data fields are of variable field widths and a delimiter character is used to separate fields, this utility can interpret the delimiters correctly.

Syntax

gl_conv is a command line command whose syntax is:

```
gl_conv source_character_set target_character_set [-ddelimiter]
    [-c fieldnum [, fieldnum...] . ] < input_file > output_file
```

Arguments

Argument	Description	Requirement
source_character_set	Name of the character set of the data to be converted. This must be one of the character set names contained in the **registry** file in the **$INFORMIXDIR/gls/cm3** directory.	Required
target_character_set	Name of the character set to which the data is to be converted. This must be one of the character set names contained in the **registry** file in the **$INFORMIXDIR/gls/cm3** directory.	Required
-d	Delimiter. Specifies the kind of delimiter used to separate different fields within the binary file. Type the ASCII character for the delimiter after the “-d”. If the character is a tab of a space, enclose the entire argument in quotation marks; for all other characters, quotation marks are not required. If you omit this argument, **gl_conv** uses the ASCII vertical bar character (|) as the default delimiter. See the **Examples** section for sample delimiter specifications.	Optional
-c	Field list. Specifies the fields whose data is to be converted, by ordinal number. (The first column is “1”.) If you omit this argument, **gl_conv** converts all fields. See the **Examples** section for a sample field list specification.	Optional
input_file	Name of the file whose data is being converted.	Required
output_file	Name of the file to which the converted data is written.	Required

Usage

Use this utility to convert binary data that is to be loaded into UNIvarchar columns in a database.

gl_conv converts the data in one or more fields of the specified input file and writes the result to the specified output file. If only some of the fields are converted, the rest of the fields are written to the output file in their original format. To see an example of an output file, see "Examples" on page 6-7.

Converting Specific Fields

You can specify which fields to convert by using the `-c` option.

Specifying the Delimiter

Binary data can contain a delimiter character to separate different fields within the binary file. The `-d` option allows you to specify the delimiter character that is used.

The delimiter character can be used within the input file as a regular text character rather than as a delimiter. If it is to be read as literal text rather than as the symbol of a delimiter, the character must be preceded by a backslash (\ ). This is common UNIX practice.

For example, if the delimiter is a colon, and one of the fields contains a colon as part of its text, the text colon must be preceded by a backslash. In this example, the name of the book is *San Francisco: Past and Present*. The backslash before the colon in the title ensures that the colon will not be read as a delimiter.

```
Smith:A.E.:San Francisco\:Past and Present:Random House:1990:
```

Important: *There must be a delimiter at the end of each line of binary data to be converted. Some databases do not put a delimiter at the end of a line when they export data. If the data you are using does not have final delimiters on each line, you must add them before you can load the data into an Informix database.*

Prerequisites

Before you can run the **gl_conv** utility, you must:

- set the **$INFORMIXDIR** environment variable as instructed for your database server.
- install ESQL/C and make it available for use by the Unicode DataBlade module user.
- set the $LD_LIBRARY_PATH environment variable to point to **$INFORMIXDIR/lib** and **$INFORMIXDIR/lib/esql**.

Examples

This section provides usage examples for the **gl_conv** utility.

1. The following table illustrates the use of the `-d` parameter.

Delimiter	Value to specify	Example
tab	The ASCII tab character	“-d “
space	The ASCII space character	“-d “
vertical bar (|)	The ASCII “ | ” character. Do not use quotation marks.	-d |
semicolon (;)	The ASCII “;” character. Do not use quotation marks.	-d;
colon (:)	The ASCII “:” character. Do not use quotation marks.	-d:

2. This example converts data in the fourth, fifth, and sixth fields of the binary file, **excel_1** and writes the converted data to the file **unicode_1**. The code set of the original file is the MS code page 1252.

   The file contains the following data, with fields delimited by a vertical line:

   ```
   %00234|Smith|John|Grange Rd.|Dublin|Ireland|
   ```

   The following command converts the data in columns 4, 5, and 6 to UNIvarchar:

   ```
   %gl_conv 1252 unicode -c4,5,6. < excel_1 > unicode_1
   ```

   Since the vertical bar is the default delimiter for **gl_conv**, there is no need to specify a delimiter.

3. This example converts data in the second, third, and sixth fields of the binary file, **excel_2**, and writes the converted data to **unicode_2**. The contents of the fields in **excel_2** is exactly the same as **excel_1**, except that its code set is DOS code page 850, and the delimiter is a tab.

   ```
   %gl_conv 850 unicode "-d" -c2,3,6. < excel_2 > unicode_2
   ```

   In this example, you must specify the delimiter because it is not the default delimiter.

4. This is a more extensive example that converts the second and third fields in the file, **cars&planes**, to Unicode. Each record in the file has three fields (INTEGER, VARCHAR, and VARCHAR) separated by a vertical bar delimiter. The file's code set is MS code page 1252.

The contents of the file are:

```
1001|Pontiac Firebird|Beechcraft Bonanza|
1002|Ford Mustang|Piper Warrior|
1003|Chevy - Camaro|Cessna Skylane RG|
1004|Dodge Viper|Mooney Ovation|
1005|Acura NSX|Commander H1|
```

The command line syntax to convert columns 2 and 3 to Unicode data and store it in the file **cars&planes.converted** is:

```
%gl_conv 1252 unicode -c2,3. < cars&planes > cars&planes.converted
```

The output file looks like this.

```
1001|0050006f006e007400690061006300200046006900720065006200690072006400|0042006500
6500630068006300720061006600740020004200 6f006e0061006e007a0061|
1002|0046006f007200640020004d007500730074006100 6e0067|005000690070006500720020 00
57006100720072006900 6f0072|
```

Appendix

Sample Tables

A

This appendix shows the SQL statements used to create and populate the tables and data used in examples in this book.

Most examples use one of these four database tables, named **books**, **authors**, **customers**, and **univartab**.

The books Table

The following statement created the **books** table:

```
CREATE TABLE books
(bookid CHAR(5),
 title  UNIvarchar,
 author VARCHAR(30),
 summaryVARCHAR(50)
);
```

The examples use data from the following rows in the **books** table:

```
INSERT INTO books
VALUES ( “10005”,
        “Le Petit Prince”::UNIvarchar,
        “Antoine de St. Exupery”,
        “A fable for children and adults”
      );

INSERT INTO books
VALUES ( “10010”,
        “Me”::UNIvarchar,
        “Katherine Hepburn”,
        “Autobiography of the legendary actress”
      );

INSERT INTO books
VALUES (“10011”,
        “Cosmos      “::UNIvarchar,
        “Carl Sagan”,
        “Companion book to the TV series”
      );

INSERT INTO books
VALUES(“10012”,
      “She”::UNIvarchar,
        ““,
        ““
       );
```

The authors Table

The following statement created the **authors** table:

```
CREATE TABLE authors
(authid     CHAR(5),
 name       VARCHAR (30),
 country    UNIvarchar,
 codes      UNIvarchar
 );
```

The examples use data from the following row in the **authors** table:

```
INSERT INTO authors
VALUES (“A0121”,
        “Antoine de St. Exupery”,
        “France”::UNIvarchar,
        “EUFR631922”::UNIvarchar
        );
```

The customers Table

The following statement created the **customers** table:

```
CREATE TABLE customers
(custid CHAR(5),
 name   LVARCHAR,
 addressLVARCHAR,
 codes  UNIvarchar
);
```

The examples use data from the following row in the **customers** table:

```
INSERT INTO customers
VALUES ( “00123”,
    “Maria Sanchez”,
    “66 San Pablo Ave., San Francisco, CA”,
    “94111SFCA00123”::UNIvarchar
);
```

The univartab Table

The following statement created the **univartab** table, a sample table used in some examples in this book:

```
CREATE TABLE univartab
(
 itemno integer,
 car    UNIvarchar,
 plane  UNIvarchar
);
```

The examples use data from the following rows in the **univartab** table:

```
INSERT INTO univartab
VALUES
(
    1001,
    “Pontiac Firebird”::UNIvarchar,
    “Beechcraft Bonanza”::UNIvarchar
);

INSERT INTO univartab (itemno, car, plane)
VALUES
(
    2001,
    “abc”::UNIvarchar,
    “Tax”::UNIvarchar
);

INSERT INTO univartab (itemno, car, plane)
VALUES
(
    2001,
    “ABC”::UNIvarchar,
    “T?x”::UNIvarchar);
```

Glossary

ASCII	Acronym for American Standard Code for Information Interchange. ASCII is a 7-bit code that assigns numeric values to characters—the letters, numbers, punctuation marks and other symbols found on a typical keyboard—and to control codes. It is the US national variant of ISO 646 and is used with most American microcomputers and many minicomputers.
ASCII character set	See *ASCII code set*.
ASCII code set	A code set based on the ASCII standard. Typically, an ASCII character set includes the 127 standard ASCII characters plus another 128 platform-specific extensions. The extensions are often used for characters in languages other than English (ç, π), foreign money symbols (¥,£), and mathematical symbols (÷, √). Also called *ASCII character set*.
ASCII file	A document file in ASCII format. Also called a *text file*.
byte ordering	Whether the most significant byte or the least significant byte comes first in a double-byte character set. The Unicode standard does not specify any order of bytes inside a Unicode value.
character	An element in a computer character set. A character can be a symbol used to represent the sounds and concepts or a language or a control character.

Examples of characters include:

- alphabets and syllabaries (for example, hiragana), which represent the sounds of a spoken language.
- ideograms, which represent words and ideas.

- special-purpose symbols, such as abbreviations (&, @, ∴, ®), mathematical symbols (±, %, ∞, π), financial symbols ($, ¢, £), punctuation and other symbols used in writing (;, —, •, ?).
- control codes that control the display of characters (“new line,” “delete,” and “shift”).
- control characters that control the interaction with auxiliary devices, such as printers (for example, “cancel”).

Symbols include letters, numbers, ideograms, punctuation marks, mathematical symbols, and so forth. Control characters, such as “new line,” “backspace,” and “shift” control the representation of the data during display and printing; others control auxiliary devices (“print”) and perform other tasks (“cancel”).

character set An ordered collection of characters.

Each character set has at least one *code set*. See *code set*.

Note: the term *character set* and *code set* are sometimes used interchangeably. However, there is a difference. A character set is a set of characters. A code set is the mapping of that character set to a set of numeric values. A character set can have more than one code set, but a code set can have only one character set associated with it.

client locale The locale used by the client application.

The *client locale* specifies the language, territory, and code set that the client application uses to perform read and write (I/O) operations on the client computer. In addition, an SQL API client uses the client locale for literal strings (end-user formats), embedded SQL statements, host variables, and data sent to or received from the database server with ESQL library functions in an ESQL source file.

coded character set See *code set*.

code page A code set. An ordered set of characters in which a numeric index (code-point value) is associated with each character. Some operating systems (DOS, IBM mainframe) use this term to represent their code sets. See *code set*.

code set	A set of numerical codes that represents a character set. For each character in a character set, there is a one-to-one mapping between that character and a bit pattern, called a *code point*. Examples of code sets are ASCII, EBCDIC, and SJIS (Shift-jis). Sometimes called *coded character set*. For example, the Japanese Shift-JIS code set maps Japanese kanji (ideograms), kana (phonetic symbols), English characters, Greek characters, and Cyrillic characters plus some symbols to an encoding that varies between 1 and 2 bytes per character. Note: a *code set* is called a *character set* in SQL standards documents and *code page* in IBM and Microsoft documents. It is also sometimes referred to as a *coded character set*.
code set order	The bit-pattern order of characters within a code set. The order of the code points in the code set determines the sort order. Alphabetical order is a code set order. Informix Dynamic Server with Universal Data Option collates data in CHAR, VARCHAR, and TEXT columns in code set order.
collation	The ordering of character data according to a set of rules defined by the code set or the locale. Sometimes referred to as *sort*. See *collation order.* Collation is discussed in detail in the *Informix Guide to GLS Functionality*.
collation order	The order in which character data is sorted. In Informix databases, data can be sorted according to rules defined by the code set or by the locale. See *code set order* and *localized order.* Collation order is sometimes called *sort order.*
database locale	The locale of the data in a database. The *database locale* specifies the language, territory, and code set that the database server needs to interpret locale-sensitive data types (NCHAR and NVARCHAR) in a particular database correctly. The database locale determines the following information for the database: ■ The code set whose characters are valid in any character column ■ The code set whose characters are valid in the names of database objects such as databases, tables, columns, and views ■ The localized order to collate data from any NCHAR and NVARCHAR columns

The client application uses the database locale (as set on the client computer) to determine whether to perform code set conversion.

double-byte	16-bit.
double-byte characters	Characters whose representations occupy between 8 and 16 bits. Languages that use ideograms, such as Chinese, Japanese, and Korean, generally use double-byte characters to represent their characters. Some code sets, such as the Japanese code set Shift-JIS, include both single-byte (kana) and double-byte (kanji) characters. Unicode is a double-byte code set.
external ASCII representation	How 16-bit data is represented in an ASCII file. Unicode characters are represented as double-byte hexadecimal values. For example, a space (hexadecimal “20”) is represented as “0020”.
field	In this product, the term *field* is used in its old-fashioned sense—a field in a record in a flat file.
GLS	See *Global Language Support*.
Global Language Support	A feature that enables Informix database products to easily handle different languages, cultural conventions, and code sets for Asian, European, Latin American, and Middle Eastern countries. See the *Informix Guide to GLS Functionality* for a complete discussion of GLS.
hexadecimal representation	A representation of data using a base-16 numeric system. The Informix Unicode DataBlade module passes Unicode data to and from the client using the LVARCHAR data type, which is displayed in DB-Access in two-byte hexadecimal format. ASCII characters, which normally have a single-byte hexadecimal representation (for example, hexadecimal “20” represents a space) are displayed as double-byte hexadecimal characters (“0020” for a space).
internal database representation	How Unicode 16-bit data is stored within the database. In the Unicode DataBlade module, Unicode data is stored in UNIvarchar columns.
locale	The language-related features of a computing environment.

A *locale* is a set of files that defines the characteristics of a particular language (for example, French), a particular territory (for example, France) and a particular code set (for example, MS Windows Code Page 1251 or ISO 8859-1). A locale specifies the code set; the collation order; end-user formats for monetary, numeric, date, and time data; and, for comparisons, the character classification and alphabetic case conversion.

See "The Unicode Locale" on page 1-6 to see how locales are used with the Unicode DataBlade module.

localized order	The order of the characters that relates to a real language. The COLLATION category of the locale file defines the order of the characters for that locale. Informix Dynamic Server with Universal Data Option collates data in NCHAR and NVARCHAR columns in code set order. UNIvarchar data is always collated in localized order.
multibyte character	Characters whose representations occupy more than 8-bits.
non-ASCII characters	All characters that are not represented in the standard ASCII code set. In this book, *non-ASCII characters* usually refers to double-byte characters. This term sometimes includes the characters in the extended ASCII code set—the characters whose code points are greater than 127.
opaque data type	A fundamental *data type* you define that contains one or more values encapsulated with an internal length and input and output functions that convert text to and from an internal storage format. Opaque types need *user-defined routines* and *user-defined operators* that work on them. Synonym for *base type*, *user-defined base type*.
server locale	The locale that the database server uses for its server-specific files. The *server locale* specifies the language, territory, and code set that the database server uses to perform read and write (I/O) operations on the server computer (the computer on which the database server runs). These I/O operations include reading or writing the following files: ■ Diagnostic files that the database server generates to provide additional diagnostic information ■ Log files that the database server generates to record events ■ Explain file, **sqexplain.out**, that the SQL statement SET EXPLAIN generates

However, the database server does not use the server locale to write files that are in an Informix-proprietary format (database and table files).

The database server is the only Informix product that needs to know the server locale. Any database server utilities that you run on the server installation use the client locale to read from and write to files and the database locale, on the server computer, to access databases that are set on the server computer.

single-byte characters	Characters that can be represented in a single byte. Most European languages have fewer than 256 characters, so their characters can be represented by 7-bit or 8-bit code sets. When an application handles data in such code sets, it can assume that 1 byte stores 1 character.
sort order	Synonym of *collation order.*
text file	See *ASCII file*.
Unicode	A standardized, cross-platform, double-byte code set that, with a few exceptions, includes every character represented in regional character sets. All characters are represented as 16-bit characters, including characters in single-byte code sets such as ASCII. Adopted as ISO 10646. (Note: Unicode represents Japanese kanji as Chinese characters and is therefore little used in Japan.)

Index