Informix Product Family
Informix
Version 12.10

IBM Informix Guide to SQL: Reference

SC27-4522-05

Informix Product Family
Informix
Version 12.10

IBM Informix Guide to SQL: Reference

SC27-4522-05

Note

Before using this information and the product it supports, read the information in "Notices" on page D-1.

Edition

This edition replaces SC27-4522-04.

Contents

Introduction . . . ix
About this publication . . . ix
Types of users . . . ix
Software compatibility . . . ix
Assumptions about your locale . . . ix
Demonstration databases . . . x
What's new in SQL Reference for Informix, Version 12.10 . . . x
Example code conventions . . . xii
Additional documentation . . . xiii
Compliance with industry standards. . . . xiii
How to read the syntax diagrams. . . . xiii
How to provide documentation feedback . . . xv

Chapter 1. System catalog tables . . . 1-1
Objects That the System Catalog Tables Track. . . . 1-1
Using the system catalog . . . 1-1
Accessing the system catalog . . . 1-6
Update system catalog data. . . . 1-6
Structure of the System Catalog . . . 1-7
SYSAGGREGATES. . . . 1-9
SYSAMS. . . . 1-10
SYSATTRTYPES . . . 1-12
SYSAUTOLOCATE . . . 1-13
SYSBLOBS . . . 1-14
SYSCASTS . . . 1-14
SYSCHECKS . . . 1-15
SYSCHECKUDRDEP . . . 1-15
SYSCOLATTRIBS. . . . 1-16
SYSCOLAUTH . . . 1-16
SYSCOLDEPEND. . . . 1-17
SYSCOLUMNS . . . 1-17
Storing column length . . . 1-20
Storing Maximum and Minimum Values . . . 1-21
SYSCONSTRAINTS . . . 1-22
SYSDEFAULTS . . . 1-22
SYSDEPEND . . . 1-23
SYSDIRECTIVES . . . 1-24
SYSDISTRIB . . . 1-24
SYSDOMAINS. . . . 1-26
SYSERRORS . . . 1-26
SYSEXTCOLS . . . 1-27
SYSEXTDFILES . . . 1-27
SYSEXTERNAL . . . 1-28
SYSFRAGAUTH . . . 1-28
SYSFRAGDIST. . . . 1-29
SYSFRAGMENTS. . . . 1-31
SYSINDEXES . . . 1-33
SYSINDICES . . . 1-35
SYSINHERITS . . . 1-37
SYSLANGAUTH . . . 1-37
SYSLOGMAP . . . 1-38
SYSOBJSTATE . . . 1-38
SYSOPCLASSES . . . 1-39
SYSPROCAUTH . . . 1-40
SYSPROCBODY . . . 1-40

SYSPROCCOLUMNS . . . 1-41
SYSPROCEDURES . . . 1-41
SYSPROCPLAN . . . 1-44
SYSREFERENCES . . . 1-44
SYSROLEAUTH . . . 1-45
SYSROUTINELANGS . . . 1-45
SYSSECLABELAUTH . . . 1-46
SYSSECLABELCOMPONENTS . . . 1-46
SYSSECLABELCOMPONENTELEMENTS . . . 1-46
SYSSECLABELNAMES . . . 1-47
SYSSECLABELS . . . 1-47
SYSSECPOLICIES. . . . 1-47
SYSSECPOLICYCOMPONENTS . . . 1-48
SYSSECPOLICYEXEMPTIONS . . . 1-48
SYSSEQUENCES . . . 1-49
SYSSURROGATEAUTH . . . 1-49
SYSSYNONYMS . . . 1-50
SYSSYNTABLE . . . 1-50
SYSTABAMDATA . . . 1-51
SYSTABAUTH . . . 1-51
SYSTABLES. . . . 1-52
SYSTRACECLASSES. . . . 1-55
SYSTRACEMSGS . . . 1-56
SYSTRIGBODY . . . 1-56
SYSTRIGGERS . . . 1-57
SYSUSERS . . . 1-58
SYSVIEWS . . . 1-58
SYSVIOLATIONS . . . 1-59
SYSXADATASOURCES . . . 1-59
SYSXASOURCETYPES . . . 1-60
SYSXTDDESC . . . 1-60
SYSXTDTYPEAUTH . . . 1-60
SYSXTDTYPES . . . 1-61
Information Schema . . . 1-62
    Generating the Information Schema Views . . . 1-63
    Accessing the Information Schema Views . . . 1-63
    Structure of the Information Schema Views . . . 1-63

Chapter 2. Data types . . . 2-1
Summary of data types . . . 2-1
ANSI to Informix data type mapping . . . 2-5
Description of Data Types . . . 2-6
    BIGINT data type . . . 2-6
    BIGSERIAL data type . . . 2-6
    BLOB data type . . . 2-7
    BOOLEAN data type . . . 2-8
    BYTE data type . . . 2-8
    CHAR(n) data type . . . 2-9
    CHARACTER(n) data type . . . 2-10
    CHARACTER VARYING(m,r) data type . . . 2-10
    CLOB data type . . . 2-11
    DATE data type . . . 2-12
    DATETIME data type . . . 2-12
    DEC data type . . . 2-15
    DECIMAL . . . 2-15
    DISTINCT data types . . . 2-17
    DOUBLE PRECISION data types . . . 2-18
    FLOAT(n) . . . 2-18
    IDSSECURITYLABEL data type . . . 2-18
    INT data type . . . 2-19
    INT8 . . . 2-19

INTEGER data type . . . 2-19
INTERVAL data type . . . 2-19
LIST(e) data type . . . 2-22
LVARCHAR(m) data type . . . 2-23
MONEY(p,s) data type . . . 2-24
MULTISET(e) data type . . . 2-25
Named ROW . . . 2-25
NCHAR(n) data type . . . 2-25
NUMERIC(p,s) data type . . . 2-26
NVARCHAR(m,r) data type . . . 2-26
OPAQUE data types . . . 2-26
REAL data type . . . 2-27
ROW data type, Named . . . 2-27
ROW data type, Unnamed . . . 2-28
SERIAL(n) data type . . . 2-29
SERIAL8(n) data type . . . 2-30
SET(e) data type . . . 2-31
SMALLFLOAT . . . 2-33
SMALLINT data type . . . 2-33
TEXT data type . . . 2-33
Unnamed ROW . . . 2-35
VARCHAR(m,r) data type . . . 2-35
Built-In Data Types . . . 2-37
Character Data Types . . . 2-37
Large-Object Data Types . . . 2-39
Time Data Types . . . 2-41
Extended Data Types . . . 2-46
Complex data types . . . 2-46
Distinct Data Types . . . 2-49
Opaque Data Types . . . 2-49
Data Type Casting and Conversion . . . 2-50
Using Built-in Casts . . . 2-50
Using User-Defined Casts . . . 2-52
Determining Which Cast to Apply . . . 2-53
Casts for distinct types . . . 2-53
What Extended Data Types Can Be Cast? . . . 2-54
Operator Precedence . . . 2-55

Chapter 3. Environment variables . . . 3-1
Types of environment variables . . . 3-1
Limitations on environment variables . . . 3-1
Using environment variables on UNIX . . . 3-2
Setting environment variables in a configuration file . . . 3-2
Setting environment variables at login time . . . 3-3
Syntax for setting environment variables . . . 3-3
Unsetting environment variables . . . 3-4
Modifying an environment-variable setting . . . 3-4
Viewing your environment-variable settings . . . 3-4
Checking environment variables with the chkenv utility . . . 3-4
Rules of precedence for environment variables . . . 3-5
Using environment variables on Windows . . . 3-6
Where to set environment variables on Windows . . . 3-6
Setting environment variables on Windows . . . 3-6
Rules of precedence for Windows environment variables . . . 3-8
Environment variables in Informix products . . . 3-8
Environment variable portal . . . 3-8
ANSIOWNER environment variable . . . 3-16
CPFIRST environment variable . . . 3-17
CMCONFIG environment variable . . . 3-17
DBACCNOIGN environment variable . . . 3-18
DBANSIWARN environment variable . . . 3-19

DBBLOBBUF environment variable . . . 3-20
DBCENTURY environment variable . . . 3-20
DBDATE environment variable . . . 3-22
DBDELIMITER environment variable . . . 3-24
DBEDIT environment variable . . . 3-25
DBFLTMASK environment variable . . . 3-25
DBLANG environment variable . . . 3-26
DBMONEY environment variable . . . 3-27
DBONPLOAD environment variable . . . 3-27
DBPATH environment variable . . . 3-28
DBPRINT environment variable . . . 3-30
DBREMOTECMD environment variable (UNIX) . . . 3-30
DBSPACETEMP environment variable . . . 3-30
DBTEMP environment variable . . . 3-32
DBTIME environment variable . . . 3-32
DBUPSPACE environment variable . . . 3-35
DEFAULT_ATTACH environment variable . . . 3-36
DELIMIDENT environment variable . . . 3-37
ENVIGNORE environment variable (UNIX) . . . 3-38
FET_BUF_SIZE environment variable . . . 3-39
IFMXMONGOAUTH environment variable . . . 3-39
IFX_DEF_TABLE_LOCKMODE environment variable . . . 3-40
IFX_DIRECTIVES environment variable . . . 3-40
IFX_EXTDIRECTIVES environment variable . . . 3-41
IFX_LARGE_PAGES environment variable . . . 3-42
IFX_LOB_XFERSIZE environment variable . . . 3-43
IFX_LONGID environment variable . . . 3-43
IFX_NETBUF_PVTPOOL_SIZE environment variable (UNIX) . . . 3-44
IFX_NETBUF_SIZE environment variable . . . 3-44
IFX_NO_SECURITY_CHECK environment variable (UNIX) . . . 3-45
IFX_NO_TIMELIMIT_WARNING environment variable . . . 3-45
IFX_NODBPROC environment variable . . . 3-45
IFX_NOT_STRICT_THOUS_SEP environment variable . . . 3-46
IFX_ONTAPE_FILE_PREFIX environment variable . . . 3-46
IFX_PAD_VARCHAR environment variable . . . 3-46
IFX_UNLOAD_EILSEQ_MODE environment variable . . . 3-47
IFX_UPDDESC environment variable . . . 3-47
IFX_XASTDCOMPLIANCE_XAEND environment variable . . . 3-48
IFX_XFER_SHMBASE environment variable . . . 3-48
IMCADMIN environment variable . . . 3-48
IMCCONFIG environment variable . . . 3-49
IMCSERVER environment variable . . . 3-49
INFORMIXC environment variable (UNIX) . . . 3-50
INFORMIXCMNAME environment variable . . . 3-50
INFORMIXCMCONUNITNAME environment variable . . . 3-50
INFORMIXCONCSMCFG environment variable . . . 3-51
INFORMIXCONRETRY environment variable . . . 3-51
INFORMIXCONTIME environment variable . . . 3-52
INFORMIXCPPMAP environment variable . . . 3-54
INFORMIXDIR environment variable . . . 3-54
INFORMIXSERVER environment variable . . . 3-54
INFORMIXSHMBASE environment variable (UNIX) . . . 3-55
INFORMIXSQLHOSTS environment variable . . . 3-55
INFORMIXSTACKSIZE environment variable . . . 3-56
INFORMIXTERM environment variable (UNIX) . . . 3-56
INF_ROLE_SEP environment variable . . . 3-57
INTERACTIVE_DESKTOP_OFF environment variable (Windows) . . . 3-57
JAR_TEMP_PATH environment variable . . . 3-58
JAVA_COMPILER environment variable . . . 3-58
JVM_MAX_HEAP_SIZE environment variable . . . 3-58
LD_LIBRARY_PATH environment variable (UNIX) . . . 3-59

LIBPATH environment variable (UNIX) . . . 3-59
NODEFDAC environment variable . . . 3-59
ONCONFIG environment variable . . . 3-60
ONINIT_STDOUT environment variable (Windows) . . . 3-60
OPTCOMPIND environment variable . . . 3-61
OPTMSG environment variable . . . 3-62
OPTOFC environment variable . . . 3-62
OPT_GOAL environment variable (UNIX) . . . 3-63
PATH environment variable . . . 3-63
PDQPRIORITY environment variable . . . 3-64
PLCONFIG environment variable . . . 3-64
PLOAD_LO_PATH environment variable . . . 3-65
PLOAD_SHMBASE environment variable . . . 3-65
PSM_ACT_LOG environment variable . . . 3-66
PSM_CATALOG_PATH environment variable . . . 3-66
PSM_DBS_POOL environment variable . . . 3-66
PSM_DEBUG environment variable. . . . 3-67
PSM_DEBUG_LOG environment variable. . . . 3-67
PSM_LOG_POOL environment variable . . . 3-67
PSORT_DBTEMP environment variable . . . 3-68
PSORT_NPROCS environment variable . . . 3-68
RTREE_COST_ADJUST_VALUE environment variable . . . 3-69
SHLIB_PATH environment variable (UNIX) . . . 3-70
SRV_FET_BUF_SIZE environment variable . . . 3-70
STMT_CACHE environment variable . . . 3-71
TERM environment variable (UNIX) . . . 3-71
TERMCAP environment variable (UNIX) . . . 3-72
TERMINFO environment variable (UNIX) . . . 3-72
THREADLIB environment variable (UNIX) . . . 3-72
TZ environment variable . . . 3-73
USETABLENAME environment variable . . . 3-73

Appendix A. The stores_demo Database . . . A-1
The stores_demo Database Map . . . A-1

Appendix B. The superstores_demo database . . . B-1
Structure of the superstores_demo Tables . . . B-1
User-defined routines and extended data types . . . B-1
Table Hierarchies . . . B-3

Appendix C. Accessibility . . . C-1
Accessibility features for IBM Informix products . . . C-1
Accessibility features . . . C-1
Keyboard navigation . . . C-1
Related accessibility information . . . C-1
IBM and accessibility . . . C-1
Dotted decimal syntax diagrams . . . C-1

Notices . . . D-1
Privacy policy considerations . . . D-3
Trademarks . . . D-3

Index . . . X-1

Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About this publication

This publication includes information about the system catalog tables, data types, and environment variables that IBM® Informix® products use.

This publication is one of a series of publications that contains information about the IBM Informix implementation of SQL. The *IBM Informix Guide to SQL: Syntax* contains all the syntax descriptions for SQL and stored procedure language (SPL). The *IBM Informix Guide to SQL: Tutorial* shows how to use basic and advanced SQL and SPL routines to access and manipulate the data in your databases. The *IBM Informix Database Design and Implementation Guide* shows how to use SQL to implement and manage your databases.

See the documentation notes files for a list of the publications in the documentation set of IBM Informix.

Types of users

This publication is written for the following users:
- Database users
- Database administrators
- Database server administrators
- Database-application programmers
- Performance engineers

This publication assumes that you have the following background:
- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming
- Some experience with database server administration, operating-system administration, or network administration

Software compatibility

For information about software compatibility, see the IBM Informix release notes.

Assumptions about your locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation and representation of numeric data, currency, date, and time that is used by a language within a given territory and encoding is brought together in a single environment, called a Global Language Support (GLS) locale.

The IBM Informix OLE DB Provider follows the ISO string formats for date, time, and money, as defined by the Microsoft OLE DB standards. You can override that default by setting an Informix environment variable or registry entry, such as GL_DATE.

If you use Simple Network Management Protocol (SNMP) in your Informix environment, note that the protocols (SNMPv1 and SNMPv2) recognize only English code sets. For more information, see the topic about GLS and SNMP in the *IBM Informix SNMP Subagent Guide*.

The examples in this publication are written with the assumption that you are using one of these locales: en_us.8859-1 (ISO 8859-1) on UNIX platforms or en_us.1252 (Microsoft 1252) in Windows environments. These locales support U.S. English format conventions for displaying and entering date, time, number, and currency values. They also support the ISO 8859-1 code set (on UNIX and Linux) or the Microsoft 1252 code set (on Windows), which includes the ASCII code set plus many 8-bit characters such as é and ñ.

You can specify another locale if you plan to use characters from other locales in your data or your SQL identifiers, or if you want to conform to other collation rules for character data.

For instructions about how to specify locales, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration databases

The DB-Access utility, which is provided with your IBM Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix publications are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases are in the `$INFORMIXDIR/bin` directory on UNIX platforms and in the `%INFORMIXDIR%\bin` directory in Windows environments.

What's new in SQL Reference for Informix, Version 12.10

This publication includes information about new features and changes in existing functionality.

The following changes and enhancements are relevant to this publication. For a complete list of what's new in this release, go to http://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.po.doc/new_features_ce.htm.

Table 1. What's new in IBM Informix Guide to SQL: Reference for Version 12.10.xC4

Overview	Reference
Information on indexes on JSON and BSON columns The new **indexattr** and **jparams** columns in the SYSINDICES system catalog table contain information about indexes on JSON and BSON columns.	"SYSINDICES" on page 1-35
Customize the display widths of Unicode private-use characters Starting in Informix GLS 6.00.xC4, you can specify the display widths that DB-Access and other character-based Informix applications use for characters in the Unicode Private Use Area (PUA) ranges. Before you try to display the characters that are in PUA ranges, set the new IFX_PUA_DISPLAY_MAPPING environment variable, and create a mapping file: `$INFORMIXDIR/gls/etc/pua.map`. In the file, list each character followed by the character representation display width. Valid display widths are 1 (halfwidth character representation) or 2 (fullwidth character representation). If you do not specify a display width for a character in the file, the default is halfwidth.	"Environment variable portal" on page 3-8

Table 2. What's new in IBM Informix Guide to SQL: Reference for Version 12.10.xC3

Overview	Reference
Automatic location and fragmentation In previous releases, the default location for new databases was the root dbspace. The default location for new tables and indexes was in the dbspace of the corresponding database. By default new tables were not fragmented. As of 12.10.xC3, you can enable the database server to automatically choose the location for new databases, tables, and indexes. The location selection is based on an algorithm that gives higher priority to non-critical dbspaces and dbspaces with an optimal page size. New tables are automatically fragmented in round-robin order in the available dbspaces. The list of available dbspaces is stored in the SYSAUTOLOCATE system catalog table.	"SYSAUTOLOCATE" on page 1-13

Table 3. What's new in IBM Informix Guide to SQL: Reference for Version 12.10.xC2

Overview	Reference
Defining separators for fractional seconds in date-time values Now you can control which separator to use in the character-string representation of fractional seconds. To define a separator between seconds and fractional seconds, you must include a literal character between the %S and %F directives when you set the **GL_DATETIME** or **DBTIME** environment variable, or when you call the TO_CHAR function. By default, a separator is not used between seconds and fractional seconds. Previously, the ASCII 46 character, a period (.), was inserted before the fractional seconds, regardless of whether the formatting string included an explicit separator for the two fields.	"DBTIME environment variable" on page 3-32

Table 4. What's new in IBM Informix Guide to SQL: Reference for Version 12.10.xC1

Overview	Reference
New CREATE TABLE and ALTER FRAGMENT syntax for rolling window tables The ROLLING FRAGMENTS and LIMIT TO options of the Interval Fragment clause can define an upper limit on the allocated storage size of any rolling window table, or on the number of fragments in a rolling window fragmentation strategy. Fragments that reach the limit can be detached from the table, and either be archived or destroyed. These specifications define a purging policy for the rolling window table. Rolling window tables require a RANGE INTERVAL storage distribution strategy and a purging policy. When a new rolling fragment is created and registered in the sysfragments system catalog table, a record in that table describes the rolling fragment and its purging policy, using new encodings that the SYSFRAGMENTS topic identifies.	"SYSFRAGMENTS" on page 1-31
Enhanced built-in storage management for backup and restore IBM Informix Primary Storage Manager, which replaces IBM Informix Storage Manager (ISM), is easier to set up and use, even in embedded environments. You use the Informix Primary Storage Manager **onpsm** utility to manage storage for ON-Bar backup and restore operations, including parallel backups, that use file devices (disks). You can use new configuration parameters and environment variables with the Informix Primary Storage Manager.	"PSM_ACT_LOG environment variable" on page 3-66 "PSM_CATALOG_PATH environment variable" on page 3-66 "PSM_DBS_POOL environment variable" on page 3-66 "PSM_DEBUG environment variable" on page 3-67 "PSM_DEBUG_LOG environment variable" on page 3-67 "PSM_LOG_POOL environment variable" on page 3-67

Example code conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept that is being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access Informix technical information such as information centers, technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

How to read the syntax diagrams

Syntax diagrams use special components to describe the syntax for SQL statements and commands.

Read the syntax diagrams from left to right and top to bottom, following the path of the line.

The double right arrowhead and line symbol ▸▸— indicates the beginning of a syntax diagram.

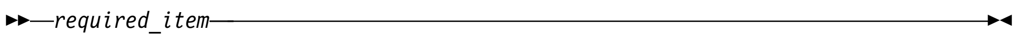
The line and single right arrowhead symbol —▸ indicates that the syntax is continued on the next line.

The right arrowhead and line symbol ►— indicates that the syntax is continued from the previous line.

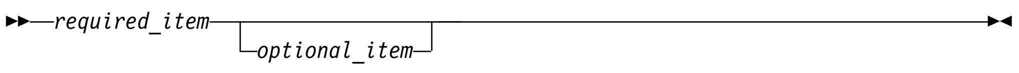
The line, right arrowhead, and left arrowhead symbol —►◄ symbol indicates the end of a syntax diagram.

Syntax fragments start with the pipe and line symbol |— and end with the —| line and pipe symbol.

Required items appear on the horizontal line (the main path).

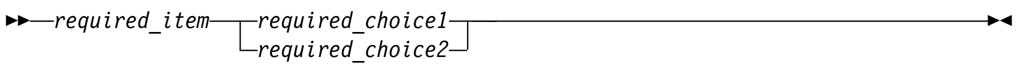


Optional items appear below the main path.

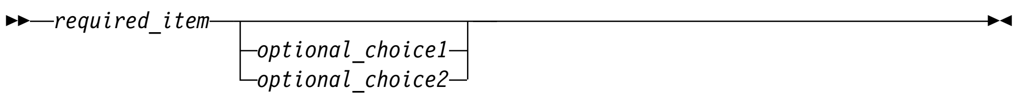


If you can choose from two or more items, they appear in a stack.

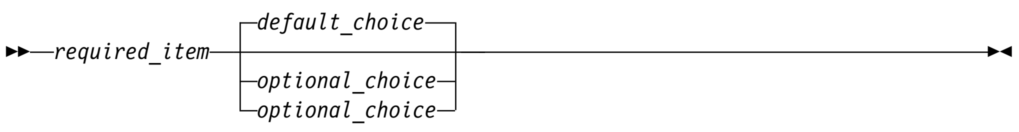
If you *must* choose one of the items, one item of the stack appears on the main path.



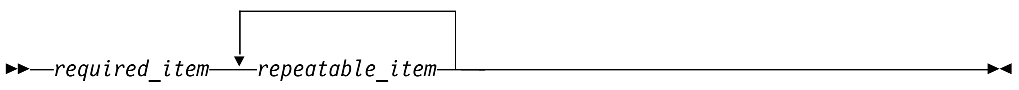
If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.



An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.



If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                  ┌─,──────────────┐
                  ▼                │
►►──required_item───repeatable_item─┴───────────────────────────────────►◄
```

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

SQL keywords appear in uppercase (for example, `FROM`). They must be spelled exactly as shown. Variables appear in lowercase (for example, `column-name`). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a syntax segment. For example, in the following diagram, the variable `parameter-block` represents the syntax segment that is labeled **parameter-block**:

```
►►──required_item──┤ parameter-block ├──────────────────────────────────►◄
```

parameter-block:

```
├──┬─parameter1──────────────────┬──────────────────────────────────────┤
   └─parameter2──┬─parameter3─┬──┘
                 └─parameter4─┘
```

How to provide documentation feedback

You are encouraged to send your comments about IBM Informix product documentation.

Add comments about documentation to topics directly in IBM Knowledge Center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback is monitored by the team that maintains the user documentation. The comments are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Software Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. System catalog tables

The *system catalog* consists of tables and views that describe the structure of the database. Sometimes called the *data dictionary*, these table objects contain everything that the database knows about itself. Each system catalog table contains information about specific elements in the database. Each database has its own system catalog.

These topics provide information about the structure, content, and use of the system catalog tables. It also contains information about the Information Schema, which provides information about the tables, views, and columns in all the databases of the IBM Informix instance to which your user session is currently connected.

Objects That the System Catalog Tables Track

The system catalog tables maintain information about the database, including the following categories of database objects:

- Tables, views, synonyms, and table fragments
- Columns, constraints, indexes, and index fragments
- Distribution statistics for tables, indexes, and fragments
- Triggers on tables, and INSTEAD OF triggers on views
- Procedures, functions, routines, and associated messages
- Authorized users, roles, and privileges to access database objects
- LBAC security policies, components, labels, and exemptions
- Data types and casts
- User-defined aggregate functions
- Access methods and operator classes
- Sequence objects
- Storage spaces for BLOB and CLOB objects
- External optimizer directives
- Inheritance relationships
- XA data sources and XA data source types
- Trusted user and surrogate user information

Using the system catalog

IBM Informix automatically generate the system catalog tables when you create a database. You can query the system catalog tables as you would query any other table in the database. The system catalog tables for a newly created database are located in a common area of the disk called a *dbspace*. Every database has its own system catalog tables. All tables and views in the system catalog have the prefix **sys** (for example, the **systables** system catalog table).

Not all tables with the prefix **sys** are true system catalog tables. For example, the **syscdr** database supports the Enterprise Replication feature. Non-catalog tables, however, have a **tabid** >= 100. System catalog tables all have a **tabid** < 100. See later in this section and "SYSTABLES" on page 1-52 for more information about **tabid** numbers that the database server assigns to tables, views, synonyms, and (in IBM Informix) sequence objects.

Tip: Do not confuse the system catalog tables of a database with the tables in the **sysmaster**, **sysutils**, **syscdr**, or (for IBM Informix) the **sysadmin** and **sysuser** databases. The names of tables in those databases also have the **sys** prefix, but they contain information about an entire database server, which might manage multiple databases. Information in the **sysadmin**, **sysmaster**, **sysutils**, **syscdr**, and **sysuser** tables is primarily useful for database server administrators (DBSAs). See also the *IBM Informix Administrator's Guide* and *IBM Informix Administrator's Reference*.

The database server accesses the system catalog constantly. Each time an SQL statement is processed, the database server accesses the system catalog to determine system privileges, add or verify table or column names, and so on.

For example, the following CREATE SCHEMA block adds the **customer** table, with its indexes and privileges, to the **stores_demo** database. This block also adds a view, **california**, which restricts the data of the **customer** table to only the first and last names of the customer, the company name, and the telephone number for all customers who reside in California.

```
CREATE SCHEMA AUTHORIZATION maryl
CREATE TABLE customer (customer_num SERIAL(101), fname CHAR(15),
   lname CHAR(15), company CHAR(20), address1 CHAR(20), address2 CHAR(20),
   city CHAR(15), state CHAR(2), zipcode CHAR(5), phone CHAR(18))
GRANT ALTER, ALL ON customer TO cathl WITH GRANT OPTION AS maryl
GRANT SELECT ON customer TO public
GRANT UPDATE (fname, lname, phone) ON customer TO nhowe
CREATE VIEW california AS
   SELECT fname, lname, company, phone FROM customer WHERE state = 'CA'
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
CREATE INDEX state_ix ON customer (state)
```

To process this CREATE SCHEMA block, the database server first accesses the system catalog to verify the following information:

- The new table and view names do not already exist in the database. (If the database is ANSI-compliant, the database server verifies that the new names do not already exist for the specified owners.)
- The user has permission to create tables and grant user privileges.
- The column names in the CREATE VIEW and CREATE INDEX statements exist in the **customer** table.

In addition to verifying this information and creating two new tables, the database server adds new rows to the following system catalog tables:

- **systables**
- **syscolumns**
- **sysviews**
- **systabauth**
- **syscolauth**
- **sysindexes**
- **sysindices**

Rows added to the systables system catalog table

The following two new rows of information are added to the **systables** system catalog table after the CREATE SCHEMA block is run.

Column name	First row	Second row
tabname	customer	california
owner	maryl	maryl
partnum	16778361	0
tabid	101	102
rowsize	134	134
ncols	10	4
nindexes	2	0
nrows	0	0
created	01/26/2007	01/26/2007
version	1	0
tabtype	T	V
locklevel	P	B
npused	0	0
fextsize	16	0
nextsize	16	0
flags	0	0
site		
dbname		

Each table recorded in the **systables** system catalog table is assigned a **tabid**, a system-assigned sequential number that uniquely identifies each table in the database. The system catalog tables receive 2-digit **tabid** numbers, and the user-created tables receive sequential **tabid** numbers that begin with `100`.

Rows added to the syscolumns system catalog table

The CREATE SCHEMA block adds 14 rows to the **syscolumns** system catalog table. These rows correspond to the columns in the table **customer** and the view **california**, as the following example shows.

colname	tabid	colno	coltype	collength	colmin	colmax
customer_num	101	1	262	4		
fname	101	2	0	15		
lname	101	3	0	15		
company	101	4	0	20		
address1	101	5	0	20		
address2	101	6	0	20		
city	101	7	0	15		
state	101	8	0	2		
zipcode	101	9	0	5		
phone	101	10	0	18		

colname	tabid	colno	coltype	collength	colmin	colmax
fname	102	1	0	15		
lname	102	2	0	15		
company	102	3	0	20		
phone	102	4	0	18		

In the **syscolumns** table, each column within a table is assigned a sequential column number, **colno**, that uniquely identifies the column within its table. In the **colno** column, the **fname** column of the **customer** table is assigned the value 2 and the **fname** column of the view **california** is assigned the value 1.

The **colmin** and **colmax** columns are empty. These columns contain values when a column is the first key (or the only key) in an index, has no NULL or duplicate values, and the UPDATE STATISTICS statement has been run.

Rows added to the sysviews system catalog table

The database server also adds rows to the **sysviews** system catalog table, whose **viewtext** column contains each line of the CREATE VIEW statement that defines the view. In that column, the **x0** that precedes the column names in the statement (for example, **x0.fname**) operates as an alias that distinguishes among the same columns that are used in a self-join.

Rows added to the systabauth system catalog table

The CREATE SCHEMA block also adds rows to the **systabauth** system catalog table. These rows correspond to the user privileges granted on **customer** and **california** tables, as the following example shows.

grantor	grantee	tabid	tabauth
maryl	public	101	su-idx--
maryl	cathl	101	SU-IDXAR
maryl	nhowe	101	--*-----
	maryl	102	SU-ID---

The **tabauth** column specifies the table-level privileges granted to users on the **customer** and **california** tables. This column uses an 8-byte pattern, such as s (Select), u (Update), * (column-level privilege), i (Insert), d (Delete), x (Index), a (Alter), and r (References), to identify the type of privilege. In this example, the user **nhowe** has column-level privileges on the **customer** table. A hyphen (-) means the user has not been granted the privilege whose position the hyphen occupies within the **tabauth** value.

If the **tabauth** privilege code is in uppercase (for example, S for Select), the user has this privilege and can also grant it to others; but if the privilege code is lowercase (for example, s for Select), the user cannot grant it to others.

Rows added to the syscolauth system catalog table

In addition, three rows are added to the **syscolauth** system catalog table. These rows correspond to the user privileges that are granted on specific columns in the **customer**, table as the following example shows.

grantor	grantee	tabid	colno	colauth
maryl	nhowe	101	2	-u-
maryl	nhowe	101	3	-u-
maryl	nhowe	101	10	-u-

The **colauth** column specifies the column-level privileges that are granted on the **customer** table. This column uses a 3-byte, pattern such as `s` (Select), `u` (Update), and `r` (References), to identify the type of privilege. For example, the user **nhowe** has Update privileges on the second column (because the **colno** value is `2`) of the **customer** table (indicated by **tabid** value of `101`).

Rows added to the sysindexes or the sysindices table

The CREATE SCHEMA block adds two rows to the **sysindexes** system catalog table (the **sysindices** table for IBM Informix). These rows correspond to the indexes created on the **customer** table, as the following example shows.

idxname	c_num_ix	state_ix
owner	maryl	maryl
tabid	101	101
idxtype	U	D
clustered		
part1	1	8
part2	0	0
part3	0	0
part4	0	0
part5	0	0
part6	0	0
part7	0	0
part8	0	0
part9	0	0
part10	0	0
part11	0	0
part12	0	0
part13	0	0
part14	0	0
part15	0	0
part16	0	0
levels		
leaves		

idxname	c_num_ix	state_ix
nunique		
clust		
idxflags		

In this table, the **idxtype** column identifies whether the created index requires unique values (U) or accepts duplicate values (D). For example, the **c_num_ix** index on the **customer.customer_num** column is unique.

Accessing the system catalog

Normal user access to the system catalog is read-only. Users with Connect or Resource privileges cannot alter the catalog, but they can access data in the system catalog tables on a read-only basis using standard SELECT statements.

For example, the following SELECT statement displays all the table names and corresponding **tabid** codes of user-created tables in the database:

```
SELECT tabname, tabid FROM systables WHERE tabid > 99
```

When you use DB-Access, only the tables that you created are displayed. To display the system catalog tables, enter the following statement:

```
SELECT tabname, tabid FROM systables WHERE tabid < 100
```

You can use the **SUBSTR** or the **SUBSTRING** function to select only part of a source string. To display the list of tables in columns, enter the following statement:

```
SELECT SUBSTR(tabname, 1, 18), tabid FROM systables
```

Although user **informix** can modify most system catalog tables, you should not update, delete, or insert any rows in them. Modifying the content of system catalog tables can affect the integrity of the database. However, you can safely use the ALTER TABLE statement to modify the size of the next extent of system catalog tables. Changing the next extent size does not affect extents that already exist.

For certain catalog tables of IBM Informix, however, it is valid to add entries to the system catalog tables. For instance, in the case of the **syserrors** system catalog table and the **systracemsgs** system catalog table, a DataBlade® module developer can directly insert entries that are in these system catalog tables.

Update system catalog data

If you use the UPDATE STATISTICS statement to update the system catalog before executing a query or other data manipulation language (DML) statement, you can ensure that the information available to the query execution optimizer is current.

In IBM Informix, the optimizer determines the most efficient strategy for executing SQL queries and other DML operations. The optimizer allows you to query the database without requiring you to consider fully which tables to search first in a join or which indexes to use. The optimizer uses information from the system catalog to determine the best query strategy.

When you delete or modify a table, the database server does not automatically update the related statistical data in the system catalog. For example, if you delete one or more rows in a table with the DELETE statement, the **nrows** column in the **systables** system catalog table, which holds the number of rows for that table, is not updated automatically.

The UPDATE STATISTICS statement causes the database server to recalculate data in the **systables**, **sysdistrib**, **syscolumns**, and **sysindices** system catalog tables, and in the **sysindexes** view. (For operations on fragmented tables where the STATLEVEL attribute is set to FRAGMENT, it also updates the **sysfragdist** and **sysfragments** system catalog tables.) After you run UPDATE STATISTICS, the **systables** system catalog table holds the correct value in the **nrows** column. If you specify MEDIUM or HIGH mode when you run UPDATE STATISTICS, the **sysdistrib** and (for fragment-level statistics) the **sysfragdist** system catalog tables hold the updated column-distribution data.

Whenever you modify a data table extensively, use the UPDATE STATISTICS statement to update data in the system catalog. For more information about the UPDATE STATISTICS statement, see the *IBM Informix Guide to SQL: Syntax*.

Structure of the System Catalog

The following system catalog tables describe the database objects in a database.

System Catalog Tables
"SYSAGGREGATES" on page 1-9
"SYSAMS" on page 1-10
"SYSATTRTYPES" on page 1-12
"SYSAUTOLOCATE" on page 1-13
"SYSBLOBS" on page 1-14
"SYSCASTS" on page 1-14
"SYSCHECKS" on page 1-15
"SYSCHECKUDRDEP" on page 1-15
"SYSCOLATTRIBS" on page 1-16
"SYSCOLAUTH" on page 1-16
"SYSCOLDEPEND" on page 1-17
"SYSCOLUMNS" on page 1-17
"SYSCONSTRAINTS" on page 1-22
"SYSDEFAULTS" on page 1-22
"SYSDEPEND" on page 1-23
"SYSDIRECTIVES" on page 1-24
"SYSDISTRIB" on page 1-24
"SYSDOMAINS" on page 1-26
"SYSERRORS" on page 1-26
"SYSEXTCOLS" on page 1-27
"SYSEXTDFILES" on page 1-27
"SYSEXTERNAL" on page 1-28
"SYSFRAGAUTH" on page 1-28
"SYSFRAGDIST" on page 1-29

System Catalog Tables
“SYSFRAGMENTS” on page 1-31
“SYSINDEXES” on page 1-33
“SYSINDICES” on page 1-35
“SYSINHERITS” on page 1-37
“SYSLANGAUTH” on page 1-37
“SYSLOGMAP” on page 1-38
“SYSOBJSTATE” on page 1-38
“SYSOPCLASSES” on page 1-39
“SYSPROCAUTH” on page 1-40
“SYSPROCBODY” on page 1-40
“SYSPROCCOLUMNS” on page 1-41
“SYSPROCEDURES” on page 1-41
“SYSPROCPLAN” on page 1-44
“SYSREFERENCES” on page 1-44
“SYSROLEAUTH” on page 1-45
“SYSROUTINELANGS” on page 1-45
“SYSSECLABELAUTH” on page 1-46
“SYSSECLABELCOMPONENTS” on page 1-46
“SYSSECLABELCOMPONENTELEMENTS” on page 1-46
“SYSSECLABELNAMES” on page 1-47
“SYSSECLABELS” on page 1-47
“SYSSECPOLICIES” on page 1-47
“SYSSECPOLICYCOMPONENTS” on page 1-48
“SYSSECPOLICYEXEMPTIONS” on page 1-48
“SYSSEQUENCES” on page 1-49
“SYSSURROGATEAUTH” on page 1-49
“SYSSYNONYMS” on page 1-50
“SYSSYNTABLE” on page 1-50
“SYSTABAMDATA” on page 1-51
“SYSTABAUTH” on page 1-51
“SYSTABLES” on page 1-52
“SYSTRACECLASSES” on page 1-55
“SYSTRACEMSGS” on page 1-56
“SYSTRIGBODY” on page 1-56
“SYSTRIGGERS” on page 1-57
“SYSUSERS” on page 1-58
“SYSVIEWS” on page 1-58
“SYSVIOLATIONS” on page 1-59
“SYSXADATASOURCES” on page 1-59
“SYSXASOURCETYPES” on page 1-60
“SYSXTDDESC” on page 1-60

System Catalog Tables
"SYSXTDTYPEAUTH" on page 1-60
"SYSXTDTYPES" on page 1-61

In case-sensitive databases that use the default database locale (U. S. English, ISO **8859-1** code set), character columns in these tables are CHAR and VARCHAR data types. For all other locales, character columns are the NLS data types, NCHAR and NVARCHAR. For information about differences in the collation order of character data types, see the *IBM Informix GLS User's Guide*. See also theChapter 2, "Data types," on page 2-1 chapter of this publication.

Character columns in databases that are not case-sensitive

In databases that are created with the NLSCASE INSENSITIVE keywords and that use the default database locale (U. S. English, ISO **8859-1** code set), character columns in system catalog tables are CHAR and VARCHAR data types, which support case-sensitive queries. For all other database locales, character column data types in the system catalog tables are the NLS data types, NCHAR and NVARCHAR, but with the following specific exceptions:

Table_name.Column_name	**Data type**
sysams.am_sptype	CHAR(3)
syscolauth.colauth	CHAR(3)
sysdefaults.class	CHAR(1)
sysfragauth.fragauth	CHAR(6)
sysinherits.class	CHAR(1)
syslangauth.langauth	CHAR(1)
sysprocauth.procauth	CHAR(1)
sysprocedures.mode	CHAR(1)
systabauth.tabauth	CHAR(9)
systriggers.event	CHAR(1)
sysxtdtypeauth.auth	CHAR(2)

In each of these columns, case-sensitive encoding can record information that utilities of the database server require in queries on those system catalog tables. In a database that is case-insensitive, queries might return incorrect results from data stored in NCHAR or NVARCHAR columns, if different attributes of database objects are encoded as different cases of the same letter. To avoid the loss of information, CHAR data types are used for the system catalog columns listed above.

SYSAGGREGATES

The **sysaggregates** system catalog table records user-defined aggregates (UDAs). The **sysaggregates** table has the following columns.

Table 1-1. SYSAGGREGATES table column descriptions

Column	Type	Explanation
name	VARCHAR(128)	Name of the aggregate

Table 1-1. SYSAGGREGATES table column descriptions (continued)

Column	Type	Explanation
owner	CHAR(32)	Name of the owner of the aggregate
aggid	SERIAL	Unique code identifying the aggregate
init_func	VARCHAR(128)	Name of initialization UDR
iter_func	VARCHAR(128)	Name of iterator UDR
combine_func	VARCHAR(128)	Name of combine UDR
final_func	VARCHAR(128)	Name of finalization UDR
handlesnulls	BOOLEAN	NULL-handling indicator: • t = handles NULLs • f = does not handle NULLs

Each user-defined aggregate has one entry in **sysaggregates** that is uniquely identified by its identifying code (the **aggid** value). Only user-defined aggregates (aggregates that are not built in) have entries in **sysaggregates**.

Both a simple index on the **aggid** column and a composite index on the **name** and **owner** columns require unique values.

SYSAMS

The **sysams** system catalog table contains information that is required for using built-in access methods and those created by the CREATE ACCESS_METHOD statement of SQL.

The **sysams** table has the following columns.

Table 1-2. SYSAMS table column descriptions

Column	Type	Explanation
am_name	VARCHAR(128, 0)	Name of the access method
am_owner	CHAR(32)	Name of the owner of the access method
am_id	INTEGER	Unique identifying code for an access method This corresponds to the **am_id** columns in the **systables**, **sysindices**, and **sysopclasses** tables.
am_type	CHAR(1)	Type of access method: P = Primary; S = Secondary
am_sptype	CHAR(3)	Types of spaces where the access method can exist: • `A` means the access method supports extspaces and sbspaces. If the access method is built in, such as a B-tree, it also supports dbspaces. • `D` or `d` means the access method supports dbspaces only. • `DS` means the access method supports dbspaces and sbspaces. • `S` or `s` means the access method supports sbspaces only. • `X` or `x` means the access method supports extspaces only. • `sx` means the access method supports sbspaces and extspaces.

Table 1-2. SYSAMS table column descriptions (continued)

Column	Type	Explanation
am_defopclass	INTEGER	Unique identifying code for default-operator class Value is the **opclassid** from the entry for this operator class in the **sysopclasses** table.
am_keyscan	INTEGER	Whether a secondary access method supports a key scan (An access method supports a key scan if it can return a key and a rowid from a call to the **am_getnext** function.) (0 = FALSE; Non-zero = TRUE)
am_unique	INTEGER	Whether a secondary access method can support unique keys (0 = FALSE; Non-zero = TRUE)
am_cluster	INTEGER	Whether a primary access method supports clustering (0 = FALSE; Non-zero = TRUE)
am_rowids	INTEGER	Whether a primary access method supports rowids (0 = FALSE; Non-zero = TRUE)
am_readwrite	INTEGER	Whether a primary access method can both read and write (0 = access method is read-only; Non-zero = access method is read/write)
am_parallel	INTEGER	Whether an access method supports parallel execution (0 = FALSE; Non-zero = TRUE)
am_costfactor	SMALLFLOAT	The value to be multiplied by the cost of a scan to normalize it to costing done for built-in access methods The scan cost is the output of the **am_scancost** function.
am_create	INTEGER	The routine specified for the AM_CREATE purpose for this access method Value = **procid** for the routine in the **sysprocedures** table.
am_drop	INTEGER	The routine specified for the AM_DROP purpose function for this access method
am_open	INTEGER	The routine specified for the AM_OPEN purpose function for this access method
am_close	INTEGER	The routine specified for the AM_CLOSE purpose function for this access method
am_insert	INTEGER	The routine specified for the AM_INSERT purpose function for this access method
am_delete	INTEGER	The routine specified for the AM_DELETE purpose function for this access method
am_update	INTEGER	The routine specified for the AM_UPDATE purpose function for this access method
am_stats	INTEGER	The routine specified for the AM_STATS purpose function for this access method
am_scancost	INTEGER	The routine specified for the AM_SCANCOST purpose function for this access method
am_check	INTEGER	The routine specified for the AM_CHECK purpose function for this access method
am_beginscan	INTEGER	The routine specified for the AM_BEGINSCAN purpose function for this access method
am_endscan	INTEGER	The routine specified for the AM_ENDSCAN purpose function for this access method

Table 1-2. SYSAMS table column descriptions (continued)

Column	Type	Explanation
am_rescan	INTEGER	The routine specified for the AM_RESCAN purpose function for this access method
am_getnext	INTEGER	The routine specified for the AM_GETNEXT purpose function for this access method
am_getbyid	INTEGER	The routine specified for the AM_GETBYID purpose function for this access method
am_build	INTEGER	The routine specified for the AM_BUILD purpose function for this access method
am_init	INTEGER	The routine specified for the AM_INIT purpose function for this access method
am_truncate	INTEGER	The routine specified for the AM_TRUNCATE purpose function for this access method
am_expr_pushdown	INTEGER	Whether parameter descriptors are supported (0 = FALSE; Non-zero = TRUE)

For each of the columns that contain a routine for a purpose function, the value is the **sysprocedures.procid** value for the corresponding routine.

A composite index on the **am_name** and **am_owner** columns in this table allows only unique values. The **am_id** column has a unique index.

For information about access method functions, see the documentation of your access method.

SYSATTRTYPES

The **sysattrtypes** system catalog table contains information about members of a complex data type. Each row of **sysattrtypes** contains information about elements of a collection data type or fields of a row data type.

The **sysattrtypes** table has the following columns.

Table 1-3. SYSATTRTYPES table column descriptions

Column	Type	Explanation
extended_id	INTEGER	Identifying code of an extended data type Value is the same as in the **sysxtdtypes** table ("SYSXTDTYPES" on page 1-61).
seqno	SMALLINT	Identifying code of an entry having **extended_id** type
levelno	SMALLINT	Position of member in collection hierarchy
parent_no	SMALLINT	Value in the **seqno** column of the complex data type that contains this member
fieldname	VARCHAR(128)	Name of the field in a row type Null for other complex data types
fieldno	SMALLINT	Field number sequentially assigned by system (from left to right within each row type)

Table 1-3. SYSATTRTYPES table column descriptions (continued)

Column	Type	Explanation
type	SMALLINT	Code for the data type See the description of **syscolumns**.**coltype** (page "SYSCOLUMNS" on page 1-17).
length	SMALLINT	Length (in bytes) of the member
xtd_type_id	INTEGER	Code identifying this data type See the description of **sysxtdtypes.extended_id** ("SYSXTDTYPES" on page 1-61).

Two indexes on the **extended_id** column and the **xtd_type_id** column allow duplicate values. A composite index on the **extended_id** and **seqno** columns allows only unique values.

SYSAUTOLOCATE

The **sysautolocate** system catalog table indicates which dbspaces are available for automatic table fragmentation.

Table 1-4. SYSAUTOLOCATE table column descriptions

Column	Type	Explanation
dbsnum	INTEGER	The ID number of the dbspace. 0 indicates multiple dbspaces.
dbsname	VARCHAR(128,0)	The name of the dbspace. An asterisk (*) indicates multiple dbspaces.
pagesize	SMALLINT	The page size of the dbspace. 0 indicates multiple page sizes.
flags	INTEGER	• 1 = On. The dbspace is available for automatic table fragmentation. • 2 = Off. The dbspace is not available for automatic table fragmentation.

You add or remove dbspace from the list of available dbspace by running the **task()** or **admin()** SQL administration API function with one of the **autolocate** database arguments.

The **sysautolocate** system catalog table does not necessarily list every dbspace. For example, if all dbspaces are available for automatic table fragmentation, the table contains one row:

```
dbsnum      dbsname       pagesize    flags
0           *             0           1
```

If all but one dbspace is available, the table contains two rows, for example:

```
dbsnum      dbsname       pagesize    flags
0           *             0           1
12          dbs12         8           2
```

If all but two dbspaces are unavailable, the table contains three rows, for example:

```
dbsnum      dbsname        pagesize     flags
0           *              0            2
12          dbs12          8            1
13          dbs13          4            1
```

Related information:

autolocate database argument: Specify dbspaces for automatic location and fragmentation (SQL administration API)

SYSBLOBS

The **sysblobs** system catalog table specifies the storage location of BYTE and TEXT column values. Its name is based on a legacy term for BYTE and TEXT columns, blobs (also known as *simple large objects*), and does not refer to the BLOB data type of IBM Informix. The **sysblobs** table contains one row for each BYTE or TEXT column, and has the following columns.

Table 1-5. SYSBLOBS table column descriptions

Column	Type	Explanation
spacename	VARCHAR(128)	Name of partition, dbspace, or family
type	CHAR(1)	Code identifying the type of storage media: M = Magnetic
tabid	INTEGER	Code identifying the table
colno	SMALLINT	Column number within its table

A composite index on **tabid** and **colno** allows only unique values.

For information about the location and size of chunks of blobspaces, dbspaces, and sbspaces for TEXT, BYTE, BLOB, and CLOB columns, see the *IBM Informix Administrator's Guide* and the *IBM Informix Administrator's Reference*.

SYSCASTS

The **syscasts** system catalog table describes the casts in the database. It contains one row for each built-in cast, each implicit cast, and each explicit cast that a user defines. The **syscasts** table has the following columns.

Table 1-6. SYSCASTS table column descriptions

Column	Type	Explanation
owner	CHAR(32)	Owner of cast (user **informix** for built-in casts and *user* name for implicit and explicit casts)
argument_type	SMALLINT	Source data type on which the cast operates
argument_xid	INTEGER	Code for the source data type specified in the **argument_type** column
result_type	SMALLINT	Code for the data type returned by the cast
result_xid	INTEGER	Data type code of the data type named in the **result_type** column
routine_name	VARCHAR(128)	Function or procedure implementing the cast
routine_owner	CHAR(32)	Name of owner of the function or procedure specified in the **routine_name** column

Table 1-6. SYSCASTS table column descriptions (continued)

Column	Type	Explanation
class	CHAR(1)	Type of cast: E = Explicit cast I = Implicit cast S = Built-in cast

If **routine_name** and **routine_owner** have NULL values, this indicates that the cast is defined without a routine. This can occur if both of the data types specified in the **argument_type** and **result_type** columns have the same length and alignment, and are passed by reference, or passed by value.

A composite index on columns **argument_type**, **argument_xid**, **result_type**, and **result_xid** allows only unique values. A composite index on columns **result_type** and **result_xid** allows duplicate values.

SYSCHECKS

The **syschecks** system catalog table describes each check constraint defined in the database. Because the **syschecks** table stores both the ASCII text and a binary encoded form of the check constraint, it contains multiple rows for each check constraint. The **syschecks** table has the following columns.

Table 1-7. SYSCHECKS table column descriptions

Column	Type	Explanation
constrid	INTEGER	Unique code identifying the constraint
type	CHAR(1)	Form in which the check constraint is stored: B = Binary encoded s = Select T = Text
seqno	SMALLINT	Line number of the check constraint
checktext	CHAR(32)	Text of the check constraint

The text in the **checktext** column associated with `B` type in the type column is in computer-readable format. To view the text associated with a particular check constraint, use the following query with the appropriate **constrid** code:

```
SELECT * FROM syschecks WHERE constrid=10 AND type='T'
```

Each check constraint described in the **syschecks** table also has its own row in the **sysconstraints** table.

A composite index on the **constrid**, **type**, and **seqno** columns allows only unique values.

SYSCHECKUDRDEP

The **syscheckudrdep** system catalog table describes each check constraint that is referenced by a user-defined routine (UDR) in the database. The **syscheckudrdep** table has the following columns.

Table 1-8. SYSCHECKUDRDEP table column descriptions

Column	Type	Explanation
udr_id	INTEGER	Unique code identifying the UDR
constraint_id	INTEGER	Unique code identifying the check constraint

Each check constraint described in the **syscheckudrdep** table also has its own row in the **sysconstraints** system catalog table, where the **constrid** column has the same value as the **constraint_id** column of **syscheckudrdep**.

A composite index on the **udr_id** and **constraint_id** columns requires that combinations of these values be unique.

SYSCOLATTRIBS

The **syscolattribs** system catalog table describes the characteristics of smart large objects, namely CLOB and BLOB data types.

It contains one row for each sbspace referenced in the PUT clause of the CREATE TABLE statement or of the ALTER TABLE statement.

Table 1-9. SYSCOLATTRIBS table column descriptions

Column	Type	Explanation
tabid	INTEGER	Code uniquely identifying the table
colno	SMALLINT	Number of the column that contains the smart large object
extentsize	INTEGER	Pages in smart-large-object extent, expressed in KB
flags	INTEGER	Integer representation of the combination (by addition) of hexadecimal values of the following parameters: • LO_NOLOG (0x00000001 = 1) = The smart large object is not logged. • LO_LOG (0x00000010 = 2) = Logging of smart large objects conforms to current log mode of the database. • LO_KEEP_LASTACCESS_TIME (0x00000100 = 4) = Keeps a record of when this column was most recently accessed by a user. • LO_NOKEEP_LASTACCESS_TIME (0x00001000 = 8) = No record is kept of when this column was most recently accessed by a user. • HI_INTEG (0x00010000= 16) = Sbspace data pages have headers and footers to detect incomplete writes and data corruption. • MODERATE_INTEG (0x00100000= 32) = Data pages have headers but no footers.
flags1	INTEGER	Reserved for future use
sbspace	VARCHAR(128)	Name of the sbspace

A composite index on the **tabid**, **colno**, and **sbspace** columns allows only unique combinations of these values.

SYSCOLAUTH

The **syscolauth** system catalog table describes each set of discretionary access privileges granted on a column. It contains one row for each set of column-level privileges that are currently granted to a user, to a role, or to the PUBLIC group on a column in the database. The **syscolauth** table has the following columns.

Column	Type	Explanation
grantor	VARCHAR(32)	Authorization identifier of the grantor
grantee	VARCHAR(32)	Authorization identifier of the grantee
tabid	INTEGER	Code uniquely identifying the table
colno	SMALLINT	Column number within the table

Column	Type	Explanation
colauth	CHAR(3)	3-byte pattern specifying column privileges: s *or* S = Select, u *or* U = Update, r *or* R = References

If the **colauth** privilege code is uppercase (for example, S for Select), a user who has this privilege can also grant it to others. If the **colauth** privilege code is lowercase (for example, s for Select), the user who has this privilege cannot grant it to others. A hyphen (**-**) indicates the absence of the privilege corresponding to that position within the **colauth** pattern.

A composite index on the **tabid**, **grantor**, **grantee**, and **colno** columns allows only unique values. A composite index on the **tabid** and **grantee** columns allows duplicate values.

SYSCOLDEPEND

The **syscoldepend** system catalog table tracks the table columns specified in check constraints and in NOT NULL constraints. Because a check constraint can involve more than one column in a table, the **syscoldepend** table can contain multiple rows for each check constraint; one row is created for each column involved in the constraint. The **syscoldepend** table has the following columns.

Column	Type	Explanation
constrid	INTEGER	Code uniquely identifying the constraint
tabid	INTEGER	Code uniquely identifying the table
colno	SMALLINT	Column number within the table

A composite index on the **constrid**, **tabid**, and **colno** columns allows only unique values. A composite index on the **tabid** and **colno** columns allows duplicate values.

See also the **syscheckudrdep** system catalog table in “SYSCHECKUDRDEP” on page 1-15, which lists every check constraint that is referenced by a user-defined routine.

See also the **sysreferences** table in “SYSREFERENCES” on page 1-44, which describes dependencies of referential constraints.

SYSCOLUMNS

The **syscolumns** system catalog table describes each column in the database.

One row exists for each column that is defined in a table or view.

Table 1-10. The SYSCOLUMNS table

Column	Type	Explanation
colname	VARCHAR(128)	Column name
tabid	INTEGER	Identifying code of table containing the column
colno	SMALLINT	Column number The system sequentially assigns this (from left to right within each table).

Table 1-10. The SYSCOLUMNS table (continued)

Column	Type	Explanation
coltype	SMALLINT	Code indicating the data type of the column: 0 = CHAR 1 = SMALLINT 2 = INTEGER 3 = FLOAT 4 = SMALLFLOAT 5 = DECIMAL 6 = SERIAL [1] 7 = DATE 8 = MONEY 9 = NULL 10 = DATETIME 11 = BYTE 12 = TEXT 13 = VARCHAR 14 = INTERVAL 15 = NCHAR 16 = NVARCHAR 17 = INT8 18 = SERIAL8 [1] 19 = SET 20 = MULTISET 21 = LIST 22 = ROW (unnamed) 23 = COLLECTION 40 = LVARCHAR fixed-length opaque types [2] 41 = BLOB, BOOLEAN, CLOB variable-length opaque types [2] 43 = LVARCHAR (client-side only) 45 = BOOLEAN 52 = BIGINT 53 = BIGSERIAL [1] 2061 = IDSSECURITYLABEL [2, 3] 4118 = ROW (named)
collength	Any of the following data types: • Integer-based • Varying-length character • Time • Fixed-point • Simple-large-object • IDSSECURITYLABEL	The value depends on the data type of the column. For some data types, the value is the column length (in bytes). See Storing Column Length for more information.
colmin	INTEGER	Minimum column length (in bytes)
colmax	INTEGER	Maximum column length (in bytes)
extended_id	INTEGER	Data type code, from the **sysxtdtypes** table, of the data type specified in the **coltype** column

Table 1-10. The SYSCOLUMNS table (continued)

Column	Type	Explanation
seclabelid	INTEGER	The label ID of the security label associated with the column if it is a protected column. NULL otherwise.
colattr	SMALLINT	**HIDDEN** 1 - Hidden column **ROWVER** 2 - Row version column **ROW_CHKSUM** 4 - Row key column **ER_CHECKVER** 8 - ER row version column **UPGRD1_COL** 16 - ER auto primary key column **UPGRD2_COL** 32 - ER auto primary key column **UPGRD3_COL** 64 - ER auto primary key column **PK_NOTNULL** 128 - NOT NULL by PRIMARY KEY

Note:

[1] In DB-Access, an offset value of 256 is always added to these **coltype** codes because DB-Access sets SERIAL, SERIAL8, and BIGSERIAL columns to NOT NULL.

[2] The built-in opaque data types do not have a unique **coltype** value. They are distinguished by the **extended_id** column in the "SYSXTDTYPES" on page 1-61 system catalog table.

[3] DISTINCT OF VARCHAR(128).

A composite index on **tabid** and **colno** allows only unique values.

The **coltype** codes can be incremented by bitmaps showing the following features of the column.

Bit Value	Significance When Bit Is Set
0x0100	NULL values are not allowed
0x0200	Value is from a host variable
0x0400	Float-to-decimal for networked database server
0x0800	DISTINCT data type
0x1000	Named ROW type
0x2000	DISTINCT type from LVARCHAR base type
0x4000	DISTINCT type from BOOLEAN base type
0x8000	Collection is processed on client system

For example, the **coltype** value 4118 for named row types is the decimal representation of the hexadecimal value 0x1016, which is the same as the hexadecimal **coltype** value for an unnamed row type (0x016), with the named-row-type bit set. The file **$INFORMIXDIR/incl/esql/sqltypes.h** contains additional information about **syscolumns.coltype** codes.

The following table lists the **coltype** values for the built-in opaque data types:

NOT NULL constraints

Similarly, the **coltype** value is incremented by 256 if the column does not allow NULL values. To determine the data type for such columns, subtract 256 from the value and evaluate the remainder, based on the possible **coltype** values. For example, if the **coltype** value is 262, subtracting 256 leaves a remainder of 6, indicating that the column has a SERIAL data type.

Storing the column data type

The database server stores the **coltype** value as bitmap, as listed in "SYSCOLUMNS" on page 1-17.

Storing column length

The **collength** column value depends on the data type of the column.

Integer-based data types

A **collength** value for a BIGINT, BIGSERIAL, DATE, INTEGER, INT8, SERIAL, SERIAL8, or SMALLINT column is machine-independent. The database server uses the following lengths for these integer-based data types of the SQL language.

Integer-based data types	Length (in bytes)
SMALLINT	2
DATE, INTEGER, and SERIAL	4
INT8 and SERIAL8	10
BIGINT and BIGSERIAL	8

Varying-length character data types

For IBM Informix columns of the LVARCHAR type, **collength** has the value of *max* from the data type declaration, or 2048 if no maximum was specified.

For VARCHAR or NVARCHAR columns, the *max_size* and *min_space* values are encoded in the **collength** column using one of these formulas:

- If the **collength** value is positive:

  collength = (*min_space* * 256) + *max_size*
- If the **collength** value is negative:

  collength + 65536 = (*min_space* * 256) + *max_size*

Time data types

As noted previously, DATE columns have a value of 4 in the **collength** column.

For columns of type DATETIME or INTERVAL, **collength** is determined using the following formula:

```
(length * 256) + (first_qualifier * 16) + last_qualifier
```

The length is the physical length of the DATETIME or INTERVAL field, and *first_qualifier* and *last_qualifier* have values that the following table shows.

Field qualifier	Value	Field qualifier	Value
YEAR	0	FRACTION(1)	11
MONTH	2	FRACTION(2)	12
DAY	4	FRACTION(3)	13
HOUR	6	FRACTION(4)	14
MINUTE	8	FRACTION(5)	15
SECOND	10		

For example, if a DATETIME YEAR TO MINUTE column has a length of `12` (such as YYYY*:*DD*:*MO*:*HH*:*MI*)*, a *first_qualifier* value of `0` (for YEAR), and a *last_qualifier* value of `8` (for MINUTE), then the **collength** value is `3080` (from `(256 * 12) + (0 * 16) + 8`).

Fixed-point data types

The **collength** value for a MONEY or DECIMAL (*p*, *s*) column can be calculated using the following formula:

```
(precision * 256) + scale
```

Simple-large-object data types

If the data type of the column is BYTE or TEXT, **collength** holds the length of the descriptor.

Storing Maximum and Minimum Values

The **colmin** and **colmax** values hold the second-smallest and second-largest data values in the column, respectively. For example, if the values in an indexed column are `1`, `2`, `3`, `4`, and `5`, the **colmin** value is `2` and the **colmax** value is `4`. Storing the second-smallest and second-largest data values lets the query optimizer make assumptions about the range of values in the column and, in turn, further refine search strategies.

The **colmin** and **colmax** columns contain values only if the column is indexed and the UPDATE STATISTICS statement has explicitly or implicitly calculated the column distribution. If you store BYTE or TEXT data in the tblspace, the **colmin** value is encoded as `-1`.

The **colmin** and **colmax** columns are valid only for data types that fit into four bytes: SMALLFLOAT, SMALLINT, INTEGER, and the first four bytes of CHAR. The values for all other noninteger column types are the initial four bytes of the maximum or minimum value, which are treated as integers.

It is better to use UPDATE STATISTICS MEDIUM than to depend on **colmin** and **colmax** values. UPDATE STATISTICS MEDIUM gives better information and is valid for all data types.

IBM Informix does not calculate **colmin** and **colmax** values for user-defined data types. These columns, however, have values for user-defined data types if a user-defined secondary access method supplies them.

SYSCONSTRAINTS

The **sysconstraints** system catalog table lists the constraints placed on the columns in each database table. An entry is also placed in the **sysindexes** system catalog table (or **sysindices** view for IBM Informix) for each unique, primary key, or referential constraint that does not already have a corresponding entry in **sysindexes** or **sysindices**. Because indexes can be shared, more than one constraint can be associated with an index. The **sysconstraints** table has the following columns.

Table 1-11. SYSCONSTRAINTS table column descriptions

Column	Type	Explanation
constrid	SERIAL	Code uniquely identifying the constraint
constrname	VARCHAR(128)	Name of the constraint
owner	VARCHAR(32)	Name of the owner of the constraint
tabid	INTEGER	Code uniquely identifying the table
constrtype	CHAR(1)	Code identifying the constraint type: • C = Check constraint • N = Not NULL • P = Primary key • R = Referential • T = Table • U = Unique
idxname	VARCHAR(128)	Name of index corresponding to constraint
collation	CHAR(32)	Collating order at the time when the constraint was created.

A composite index on the **constrname** and **owner** columns allows only unique values. An index on the **tabid** column allows duplicate values, and an index on the **constrid** column allows only unique values.

For check constraints (where **constrtype** = `C`), the **idxname** is always NULL. Additional information about each check constraint is contained in the **syschecks** and **syscoldepend** system catalog tables.

SYSDEFAULTS

The **sysdefaults** system catalog table lists the user-defined defaults that are placed on each column in the database. One row exists for each user-defined default value.

The **sysdefaults** table has the following columns:

Table 1-12. SYSDEFAULTS table column descriptions

Column	Type	Explanation
tabid	INTEGER	Code uniquely identifying a table. When the **class** column contains the code P, then the **tabid** column references a procedure ID not a table ID.
colno	SMALLINT	Code uniquely identifying a column.
type	CHAR(1)	Code identifying the type of default value: C = Current® L = Literal value N = NULL S = Dbservername *or* Sitename T = Today U = User
default	CHAR(256)	If **sysdefaults.type** = L, a literal default value.
class	CHAR(1)	Code identifying what kind of column: T = table t = ROW type P = procedure

If no default is specified explicitly in the CREATE TABLE or the ALTER TABLE statement, then no entry exists for that column in the **sysdefaults** table.

If you specify a literal for the default value, it is stored in the **default** column as ASCII text. If the literal value is not of one of the data types listed in the next paragraph, the **default** column consists of two parts. The first part is the 6-bit representation of the binary value of the default value structure. The second part is the default value in ASCII text. A blank space separates the two parts.

If the data type of the column is not CHAR, NCHAR, NVARCHAR, or VARCHAR, or (for IBM Informix) BOOLEAN or LVARCHAR, a binary representation of the default value is encoded in the **default** column.

A composite index on the **tabid**, **colno**, and **class** columns allows only unique values.

SYSDEPEND

The **sysdepend** system catalog table describes how each view or table depends on other views or tables. One row exists in this table for each dependency, so a view based on three tables has three rows. The **sysdepend** table has the following columns.

Table 1-13. SYSDEPEND table column descriptions

Column	Type	Explanation
btabid	INTEGER	Code uniquely identifying the base table or view
btype	CHAR(1)	Base object type: T = Table V = View
dtabid	INTEGER	Code uniquely identifying a dependent table or view
dtype	CHAR(1)	Code for the type of dependent object; currently, only view (V = View) is implemented

The **btabid** and **dtabid** columns are indexed and allow duplicate values.

SYSDIRECTIVES

The **sysdirectives** table stores external optimizer directives that can be applied to queries. Whether queries in client applications can use these optimizer directives depends on the setting of the **IFX_EXTDIRECTIVES** environment variable on the client system, as described in Chapter 3, and on the EXT_DIRECTIVES setting in the configuration file of the database server.

The **sysdirectives** table has the following columns:

Table 1-14. SYSDIRECTIVES table column descriptions

Column	Type	Explanation
id	SERIAL	Unique code identifying the optimizer directive
query	TEXT	Text of the query as it exists in the application
directives	TEXT	Text of the optimizer directive, without comments
directive_code	BYTE	Encoded directive
active	SMALLINT	Integer code that identifies whether this entry is active (= 1) or test only (= 2)
hash_code	SMALLINT	For internal use only

NULL values are not valid in the **query** column. There is a unique index on the **id** column.

SYSDISTRIB

The **sysdistrib** system catalog table stores data-distribution information for the query optimizer to use. Data distributions provide detailed table and column information to the optimizer to improve the choice of execution paths of SELECT statements.

The **sysdistrib** table has the following columns.

Table 1-15. SYSDISTRIB table column descriptions

Column	Type	Explanation
tabid	INTEGER	Code identifying the table from which data values were gathered
colno	SMALLINT	Column number in the source table
seqno	INTEGER	Ordinal number for multiple entries
constructed	DATETIME YEAR TO FRACTION(5)	Date when the data distribution was created
mode	CHAR(1)	Optimization level: M = Medium H = High
resolution	SMALLFLOAT	Specified in the UPDATE STATISTICS statement
confidence	SMALLFLOAT	Specified in the UPDATE STATISTICS statement
encdat	STAT	Statistics information

Table 1-15. SYSDISTRIB table column descriptions (continued)

Column	Type	Explanation
type	CHAR(1)	Type of statistics: A = **encdat** has ASCII-encoded histogram in fixed-length character field S = **encdat** has user-defined statistics
smplsize	SMALLFLOAT	A value greater than zero up to 1.0 indicating a proportion of the total rows in the table that UPDATE STATISTICS samples. Values greater than 1.0 indicate the actual number of rows used that UPDATE STATISTICS samples. A value of zero indicates that no sample size is specified. UPDATE STATISTICS HIGH always updates statistics for all rows.
rowssmpld	FLOAT	Number of rows in the sample
constr_time	DATETIME YEAR TO FRACTION(5)	Time when the distribution was recorded
ustnrows	FLOAT	Rows in fragment when distribution was calculated.
ustbuildduration	INTERVAL HOUR TO FRACTION(5)	Time spent calculating the distribution statistics for this column
nupdates	FLOAT	Number of updates to the table
ndeletes	FLOAT	Number of deletes to the table
ninserts	FLOAT	Number of inserts to the table

Information is stored in the **sysdistrib** table when an UPDATE STATISTICS statement with mode MEDIUM or HIGH is executed for a table. (UPDATE STATISTICS LOW does not insert a value into the **mode** column.)

Only user **informix** can select the **encdat** column.

Each row in the **sysdistrib** system catalog table is keyed by the **tabid** and **colno** for which the statistics are collected.

For built-in data type columns, the **type** field is set to `A`. The **encdat** column stores an ASCII-encoded histogram that is broken down into multiple rows, each of which contains 256 bytes.

In IBM Informix, for columns of user-defined data types, the **type** field is set to S. The **encdat** column stores the statistics collected by the **statcollect** user-defined routine in multirepresentational form. Only one row is stored for each **tabid** and **colno** pair. A composite index on the **tabid**, **colno**, and **seqno** columns requires unique combinations of values.

The following three DML counter columns record counts of how many DML operations modifying data rows were performed on the table at the time of generation of column distribution statistics:

- UPDATE operations in **nupdates**
- DELETE operations in **ndeletes**
- and INSERT operations in **ninserts**

These counts can also include rows modified by MERGE statements.

These DML counter columns store the values of the counters from the server partition that exists when distribution statistics are generated. If the AUTO_STAT_MODE configuration parameter, or the AUTO_STAT_MODE session environment setting, or the AUTO keyword of the UPDATE STATISTICS statement has enabled selective updating of data distribution statistics, the **ninserts**, **ndeletes**, and **ninserts** values can affect whether UPDATE STATISTICS operations refresh existing data distribution statistics. When the UPDATE STATISTICS statement runs in MEDIUM or HIGH mode against the table, the database server compares the stored values in these columns with the current values in the partition. Column distribution statistics for the table are not updated if the sum of the stored values differs from the sum of these current **sysdistrib** DML counter values from the partition page by less than the threshold specified by the setting of the STATCHANGE table attribute or of the STATCHANGE configuration parameter.

SYSDOMAINS

The **sysdomains** view is not used. It displays columns of other system catalog tables. It has the following columns.

Table 1-16. SYSDOMAINS table column descriptions

Column	Type	Explanation
id	SERIAL	Unique code identifying the domain
owner	CHAR(32)	Name of the owner of the domain
name	VARCHAR(128)	Name of the domain
type	SMALLINT	Code identifying the type of domain

There is no index on this view.

SYSERRORS

The **syserrors** system catalog table stores information about error, warning, and informational messages returned by DataBlade modules and user-defined routines using the **mi_db_error_raise( )** DataBlade API function.

For a description of an error message, use the **finderr** utility or go to http://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.em.doc/errors.html.

The **syserrors** table has the following columns.

Column	Type	Explanation
sqlstate	CHAR(5)	SQLSTATE value associated with the error.
locale	CHAR(36)	The locale with which this version of the message is associated (for example, **en_us.8859-1**)
level	SMALLINT	Reserved for future use
seqno	SMALLINT	Reserved for future use
message	VARCHAR(255)	Message text

To create a new message, insert a row directly into the **syserrors** table. By default, all users can view this table, but only users with the DBA privilege can modify it.

A composite index on the **sqlstate**, **locale**, **level**, and **seqno** columns allows only unique values.

Related information:

Using the SQLSTATE Error Status Code

SYSEXTCOLS

The **sysextcols** system catalog table contains a row that describes each of the internal columns in external table **tabid** of format type (**fmttype**) FIXED.

The **sysextcols** table has the following columns.

Column	Type	Explanation
tabid	INTEGER	Unique identifying code of a table
colno	SMALLINT	Code identifying the column
exttype	SMALLINT	Code identifying an external column type
extstart	SMALLINT	Starting position of column in the external data file
extlength	SMALLINT	External column length (in bytes)
nullstr	CHAR(256)	Represents NULL in external data
decprec	SMALLINT	Precision for external decimals
extstype	VARCHAR(128,0)	External type name

No entries are stored in **sysextcols** for DELIMITED or IBM Informix format external files.

You can use the DBSCHEMA utility to write out the description of the external tables. To query these system catalog tables about an external table, use the **tabid** as stored in **systables** with **tabtype** = ‘E'.

An index on the **tabid** column allows duplicate values.

SYSEXTDFILES

The **sysextdfiles** system catalog table contains identifying codes and the paths of external tables.

For each external table, at least one row exists in the **sysextdfiles** system catalog table, which has the following columns.

Column	Type	Explanation
tabid	INTEGER	Unique identifying code of an external table
dfentry	CHAR(469)	Absolute source or target file path
blobdir	CHAR(344)	Absolute or relative directory name
clobdir	CHAR(344)	Absolute or relative directory name

You can use DBSCHEMA to write out the description of the external tables. To query these system catalog tables about an external table, use the **tabid** as stored in **systables** with **tabtype** = 'E'.

An index on the **tabid** column allows duplicate values.

SYSEXTERNAL

For each external table, a single row exists in the **sysexternal** system catalog table.

The **tabid** column associates the external table record in this system catalog table with an entry in **systables**.

Column	Type	Explanation
tabid	INTEGER	Unique identifying code of an external table
fmttype	CHAR(1)	Type of format: D = (delimited) F = (fixed) I = (IBM Informix)
codeset	VARCHAR(128)	Reserved for future use
recdelim	VARCHAR(128)	The record delimiter
flddelim	CHAR(4)	The field delimiter
datefmt	CHAR(8)	Reserved for future use
moneyfmt	CHAR(20)	Reserved for future use
maxerrors	INTEGER	Number of errors to allow
rejectfile	CHAR(464)	Name of the reject file
flags	INTEGER	Optional **load** flags
ndfiles	INTEGER	Number of data files in **sysextdfiles**

You can use the **dbschema** utility to write out the description of the external tables. To query these system catalog tables about an external table, use the **tabid** as stored in **systables** with **tabtype** = 'E'.

An index on the **tabid** column allows only unique values.

SYSFRAGAUTH

The **sysfragauth** system catalog table stores information about the privileges that are granted on table fragments. This table has the following columns.

Table 1-17. SYSFRAGAUTH table column descriptions

Column	Type	Explanation
grantor	CHAR(32)	Name of the grantor of privilege
grantee	CHAR(32)	Name of the grantee of privilege
tabid	INTEGER	Identifying code of the fragmented table
fragment	VARCHAR(128)	Name of dbspace where fragment is stored

Table 1-17. SYSFRAGAUTH table column descriptions (continued)

Column	Type	Explanation
fragauth	CHAR(6)	A 6-byte pattern specifying fragment privileges (including 3 bytes reserved for future use): • u *or* U = Update • i *or* I = Insert • d *or* D = Delete

In the **fragauth** column, an uppercase code (such as `U` for Update) means that the grantee can grant the privilege to other users; a lowercase (for example, `u` for Update) means the user cannot grant the privilege to others. Hyphen (-) indicates the absence of the privilege for that position within the pattern.

A composite index on the **tabid**, **grantor**, **grantee**, and **fragment** columns allows only unique values. A composite index on the **tabid** and **grantee** columns allows duplicate values.

The following example displays the fragment-level privileges for one base table, as they exist in the **sysfragauth** table. In this example, the grantee **rajesh** can grant the Update, Delete, and Insert privileges to other users.

grantor	grantee	tabid	fragment	fragauth
dba	omar	101	dbsp1	-ui---
dba	jane	101	dbsp3	--i---
dba	maria	101	dbsp4	--id--
dba	rajesh	101	dbsp2	-UID--

SYSFRAGDIST

The **sysfragdist** system catalog table stores fragment-level column statistics for fragmented tables and indexes. One row exists for each table fragment or index fragment.

Only columns in fragmented tables are described here. (For table-level column statistics, see the **sysdistrib** system catalog table.)

The **sysfragdist** table has the following columns.

Column	Type	Explanation
tabid	INTEGER	Unique identifying code of table (= **systables.tabid**)
fragid	INTEGER	Unique identifying code of fragment (= **sysfragments.partnum**)
colno	SMALLINT	Unique identifying code of column (= **syscolumns.colno**)
seqno	SMALLINT	Sequence number (for distributions that span multiple rows)
mode	CHAR(1)	UPDATE STATISTICS mode (H = high, or M = medium)

Column	Type	Explanation
resolution	SMALLFLOAT	Average percentage of the sample in each bin
confidence	SMALLFLOAT	Estimated likelihood that a MEDIUM mode sample value is equivalent to an exact HIGH mode result
rowssampled	FLOAT	Number of rows in the sample
ustbuildduration	INTERVAL HOUR TO FRACTION(5)	Time spent to calculate the distribution for this column
constr_time	DATETIME YEAR TO FRACTION(5)	Time when the distribution was recorded
ustnrows	FLOAT	Rows in fragment when distribution was calculated.
minibinsize	FLOAT	For internal use only
nupdates	FLOAT	Number of updates to the table
ndeletes	FLOAT	Number of deletes to the table
ninserts	FLOAT	Number of inserts to the table
version	INTEGER	Reserved for future use
dbsnum	INTEGER	Unique identifying code of sbspace where **encdist** is stored
encdist	STAT	Encrypted fragment distribution

The set of rows with a given combination of **tabid**, **fragid**, and **colno** values identifies the column statistics for that fragment of a table. These statistics can span multiple rows by using the **seqno** column for sequence numbering.

The *mode*, *resolution* and *confidence* values that are specified in the UPDATE STATISTICS MEDIUM or HIGH statement that calculate the column statistics for the fragment are recorded in the **sysfragdist** columns of the same names. To use existing fragment statistics to build table statistics, these three parameters should not change between UPDATE STATISTICS statements that reference the fragments of the same table. The only exception to this is that "H" mode fragmented statistics can be used to build "M" mode table statistics.

Column distribution statistics for the fragment are stored in the column **encdist**. The **dbsnum** column stores the identifying code of the smart blob space where the **encdist** object describing this fragment is stored. By default, the SBSPACENAME configuration parameter setting is the identifier of the sbspace whose identifying code is in the **dbsnum** column.

The following three columns record counts of how many DML operations modifying data rows were performed on the fragment at the time of generation of column distribution statistics:
- UPDATE operations in **nupdates**
- DELETE operations in **ndeletes**
- and INSERT operations in **ninserts**

These counts can also include rows modified by MERGE statements.

These DML counter columns store the values of the counters from the server partition that existed when distribution statistics were generated. When UPDATE

STATISTICS runs in MEDIUM or HIGH mode against the fragmented table with fragment level statistics, the database server compares the stored values in these columns with the current values in the partition.

When the AUTO_STAT_MODE configuration parameter, or the AUTO_STAT_MODE session environment setting, or the AUTO keyword of the UPDATE STATISTICS statement has enabled selective updating of data distribution statistics, the **ninserts**, **ndeletes**, and **ninserts** values can affect whether UPDATE STATISTICS operations refresh existing data distribution statistics for the fragment. Column statistics for the fragment corresponding to the row in the **sysfragdist** table are not updated if the sum of the stored values differs from the sum of these current DML counter values for the partition page by less than the threshold specified by the setting of the STATCHANGE table attribute or of the STATCHANGE configuration parameter.

SYSFRAGMENTS

The **sysfragments** system catalog table stores fragmentation information and LOW mode statistical distributions for individual fragments of tables and indexes. One row exists for each table fragment or index fragment.

The **sysfragments** table has the following columns.

Column	Type	Explanation
fragtype	CHAR(1)	Code indicating the type of fragmented object: • I = Original index fragment • T = Original table fragment
tabid	INTEGER	Unique identifying code of table
indexname	VARCHAR(128)	Name of index
colno	INTEGER	Identifying code of TEXT or BYTE column, or the upper limit on the number of rolling window fragments
partn	INTEGER	Identifying code of physical storage location
strategy	CHAR(1)	Code for type of fragment distribution strategy: • R = Round-robin distribution strategy • E = Expression-based distribution strategy • I = IN DBSPACE clause specifies a storage location as part of distribution strategy • N = raNge-iNterval (or rolliNg wiNdow) distribution strategy • L = List distribution strategy • T = Table-based distribution strategy • H = table is a subtable within a table Hierarchy
location	CHAR(1)	Reserved for future use; shows L for local
servername	VARCHAR(128)	Reserved for future use

Column	Type	Explanation
evalpos	INTEGER	Position of fragment in the fragmentation list. For fragmentation by INTERVAL, one of the following values that indicates the type of information in the **exprtext** field: • -1 = List of dbspaces for interval fragments • -2 = Interval value • -3 = Fragmentation key • -4 = Rolling window fragment Fragmentation by LIST also uses the -3 value.
exprtext	TEXT	Expression for fragmentation strategy For fragmentation by INTERVAL, LIST, or rolling window, provides the information corresponding to the value of the **evalpos** field.
exprbin	BYTE	Binary version of expression
exprarr	BYTE	Range-partitioning data to optimize expression in range-expression fragmentation strategy
flags	INTEGER	Used internally
dbspace	VARCHAR(128)	Name of dbspace storing this fragment
levels	SMALLINT	Number of B-tree index levels
npused	FLOAT	For table-fragmentation strategies: the number of data pages For index-fragmentation strategies: the number of leaf pages For rolling window tables: the units for the storage size limit in nrows
nrows	FLOAT	For tables: the number of rows in the fragment. For indexes: the number of unique keys. For rolling window tables: the upper limit on storage size in the purge policy.
clust	FLOAT	Degree of index clustering; smaller numbers correspond to greater clustering.
partition	VARCHAR(128)	Fragment name.This can match the name of the dbspace that stores the fragment, or can be an arbitrary name.
version	SMALLINT	Number that increments when fragment statistics is updated
nupdates	FLOAT	Number of updates to the fragment
ndeletes	FLOAT	Number of deletes to the fragment
ninserts	FLOAT	Number of inserts to the fragment

Every fragment has a row in this table. The **evalpos** and **evaltext** fields contain information about individual fragments.

Tables and indexes created with fragmentation by INTERVAL or LIST have additional rows containing information about the fragmentation strategy.

The **strategy** type T is used for attached indexes. (This is a fragmented index whose fragmentation strategy is the same as for the table fragmentation.)

For information about the **nupdates**, **ndeletes**, and **ninserts** columns, which in **sysfragments** tabulate DML operations on a table since the most recent recalculation of its distribution statistics, see the description of the three columns that have the same names in the "SYSDISTRIB" on page 1-24 system catalog table.

In Informix, a composite index on the **fragtype**, **tabid**, **indexname**, and **evalpos** columns allows duplicate values.

SYSINDEXES

The **sysindexes** table is a view on the **sysindices** table. It contains one row for each index in the database.

The **sysindexes** table has the following columns.

Table 1-18. SYSINDEXES table column descriptions

Column	Type	Explanation
idxname	VARCHAR(128)	Index name
owner	VARCHAR(32)	Owner of index (user **informix** for system catalog tables and *username* for database tables)
tabid	INTEGER	Unique identifying code of table
idxtype	CHAR(1)	Index type: U = Unique; D = Duplicates allowed; G = Nonbitmap generalized-key index; g = Bitmap generalized-key index; u = unique, bitmap; d = nonunique, bitmap
clustered	CHAR(1)	Clustered or nonclustered index (C = Clustered)
part1	SMALLINT	Column number (**colno**) of a single index or the 1st component of a composite index
part2	SMALLINT	2nd component of a composite index
part3	SMALLINT	3rd component of a composite index
part4	SMALLINT	4th component of a composite index
part5	SMALLINT	5th component of a composite index
part6	SMALLINT	6th component of a composite index

Table 1-18. SYSINDEXES table column descriptions (continued)

Column	Type	Explanation
part7	SMALLINT	7th component of a composite index
part8	SMALLINT	8th component of a composite index
part9	SMALLINT	9th component of a composite index
part10	SMALLINT	10th component of a composite index
part11	SMALLINT	11th component of a composite index
part12	SMALLINT	12th component of a composite index
part13	SMALLINT	13th component of a composite index
part14	SMALLINT	14th component of a composite index
part15	SMALLINT	15th component of a composite index
part16	SMALLINT	16th component of a composite index
levels	SMALLINT	Number of B-tree levels
leaves	INTEGER	Number of leaves
nunique	INTEGER	Number of unique keys in the first column
clust	INTEGER	Degree of clustering; smaller numbers correspond to greater clustering
idxflags	INTEGER	Bitmap storing the current locking mode of the index

As with most system catalog tables, changes that affect existing indexes are reflected in this table only after you run the UPDATE STATISTICS statement.

Each **part1** through **part16** column in this table holds the column number (**colno**) of one of the 16 possible parts of a composite index. If the component is ordered in descending order, the **colno** is entered as a negative value. The columns are filled in for B-tree indexes that do not use user-defined data types or functional indexes. For generic B-trees and all other access methods, the **part1** through **part16** columns all contain zeros.

The **clust** column is blank until the UPDATE STATISTICS statement is run on the table. The maximum value is the number of rows in the table, and the minimum value is the number of data pages in the table.

SYSINDICES

The **sysindices** system catalog table describes the indexes in the database. It stores LOW mode statistics for all indexes, and contains one row for each index that is defined in the database.

Table 1-19. **sysindices** *system catalog table columns*

Column	Type	Explanation
idxname	VARCHAR(128)	Name of index
owner	VARCHAR(32)	Name of owner of index (user **informix** for system catalog tables and *username* for database tables)
tabid	INTEGER	Unique identifying code of table

idxtype	CHAR(1)	Uniqueness status
		U = Unique values required
		D = Duplicates allowed

clustered	CHAR(1)	Clustered or nonclustered status (C = Clustered)
levels	SMALLINT	Number of tree levels
leaves	FLOAT	Number of leaves
nunique	FLOAT	Number of unique keys in the first column
clust	FLOAT	Degree of clustering; smaller numbers correspond to greater clustering. The maximum value is the number of rows in the table, and the minimum value is the number of data pages in the table. This column is blank until UPDATE STATISTICS is run on the table.
nrows	FLOAT	Estimated number of rows in the table (zero until UPDATE STATISTICS is run on the table)
indexkeys	INDEXKEYARRAY	Internal representation of the index keys. Column can have up to three fields, in the format: **procid, (***col1*,*col2*, . . . , *coln***), opclassid** where 1 < *n* < 341
amid	INTEGER	Unique identifying code of the access method that implements this index. (Value = **am_id** for that access method in the **sysams** table.)

Table 1-19. ***sysindices*** *system catalog table columns (continued)*

Column	Type	Explanation
amparam	LVARCHAR(2048)	List of parameters used to customize the **amid** access method behavior
collation	CHAR(32)	Database locale whose collating order was in effect at the time of index creation
pagesize	INTEGER	Size of the page, in bytes, where this index is stored
nhashcols	SMALLINT	Number of hashed columns in a FOT index
nbuckets	SMALLINT	Number of subtrees (buckets) in a forest of trees (FOT) index
ustlowts	DATETIME YEAR TO FRACTION	Date and time when index statistics were last recorded
ustbuildduration	INTERVAL HOUR TO FRACTION(5)	Time required to calculate index statistics
nupdates	FLOAT	Number of updates to the table
ndeletes	FLOAT	Number of deletes to the table
ninserts	FLOAT	Number of inserts to the table
fextsize	INT	Size (in KB) of the first extent of the index
nextsize	INT	Size (in KB) of the next extent of the index
indexattr	INT	• 0x00000001 = The index has a partial column key • 0x00000002 = The index is compressed • 0x00000004 = The index is on a BSON column
jparam	LVARCHAR(2048)	BSON index information

Tip: This system catalog table is changed from Version 7.2 of IBM Informix. The earlier schema of this system catalog table is still available as a view that can be accessed under its original name: **sysindexes**. See “SYSINDEXES” on page 1-33.

Changes that affect existing indexes are reflected in this system catalog table only after you run the UPDATE STATISTICS statement.

The fields within the **indexkeys** columns have the following significance:
- The **procid** (as in **sysprocedures**) exists only for a functional index on return values of a function defined on columns of the table.
- The list of columns (*col1*, *col2*, ... , *coln*) in the second field identifies the columns on which the index is defined. The maximum is language-dependent: up to 341 for an SPL or Java™ UDR; up to 102 for a C UDR.

- The **opclassid** identifies the secondary access method that the database server used to build and to search the index. This is the same as the **sysopclasses.opclassid** value for the access method.

For information about the **nupdates**, **ndeletes**, and **ninserts** columns, which in **sysindices** tabulate DML operations on an index since the most recent recalculation of its distribution statistics, see the description of the three columns that have the same names in the "SYSDISTRIB" on page 1-24 system catalog table.

The **fextsize** column shows the user-defined first extent size (in kilobytes) that the optional EXTENT SIZE clause specified in the CREATE INDEX statement that defined the index. Similarly, the **nextsize** column shows the user-defined next extent size (in kilobytes) that the optional NEXT SIZE clause specified in the CREATE INDEX statement. Each of these columns displays a value of zero (0) if the corresponding EXTENT SIZE or NEXT SIZE clause was omitted when the index was created.

If the CREATE INDEX statement that defines a new index includes no explicit extent size specifications, the database server automatically calculates the first and next extent sizes, but the **fextsize** and **nextsize** column values are set to 0. When the database server is converted from a release earlier than Version 11.70, the **fextsize** and **nextsize** values for every migrated index are 0.

The **tabid** column is indexed and allows duplicate values. A composite index on the **idxname**, **owner**, and **tabid** columns allows only unique values.

SYSINHERITS

The **sysinherits** system catalog table stores information about table hierarchies and named ROW type inheritance. Every supertype, subtype, supertable, and subtable in the database has a corresponding row in the **sysinherits** table.

Column	Type	Explanation
child	INTEGER	Identifying code of the subtable or subtype
parent	INTEGER	Identifying code of the supertable or supertype
class	CHAR(1)	Inheritance class: t = named ROW type T = table

The **child** and **parent** values are from **sysxtdtypes.extended_id** for named ROW types, or from **systables.tabid** for tables. Simple indexes on the **child** and **parent** columns allow duplicate values.

SYSLANGAUTH

The **syslangauth** system catalog table contains the authorization information about computer languages that are used to write user-defined routines (UDRs).

Table 1-20. SYSLANGAUTH table column descriptions

Column	Type	Explanation
grantor	VARCHAR(32)	Name of the grantor of the language authorization
grantee	VARCHAR(32)	Name of the grantee of the language authorization

Table 1-20. SYSLANGAUTH table column descriptions (continued)

Column	Type	Explanation
langid	INTEGER	Identifying code of language in **sysroutinelangs** table
langauth	CHAR(1)	The language authorization: u = Usage privilege granted; U = Usage privilege granted WITH GRANT OPTION

A composite index on the **langid**, **grantor**, and **grantee** columns allows only unique values. A composite index on the **langid** and **grantee** columns allows duplicate values.

SYSLOGMAP

The **syslogmap** system catalog table contains fragmentation information.

Table 1-21. SYSLOGMAP table column descriptions

Column	Type	Explanation
tabloc	INTEGER	Code for the location of a table in another database
tabid	INTEGER	Unique identifying code of the table
fragid	INTEGER	Identifying code of the fragment
flags	INTEGER	Bitmap of modifiers from declaration of fragment

A simple index on the **tabloc** column and a composite index on the **tabid** and **fragid** columns do not allow duplicate values.

SYSOBJSTATE

The **sysobjstate** system catalog table stores information about the state (object mode) of database objects. The types of database objects that are listed in this table are indexes, triggers, and constraints.

Every index, trigger, and constraint in the database has a corresponding row in the **sysobjstate** table if a user creates the object. Indexes that the database server creates on the system catalog tables are not listed in the **sysobjstate** table because their object mode cannot be changed.

The **sysobjstate** table has the following columns.

Table 1-22. SYSOBJSTATE table column descriptions

Column	Type	Explanation
objtype	CHAR(1)	Code for the type of database object: • C = Constraint • I = Index • T = Trigger
owner	VARCHAR(32)	Authorization identifier of the owner of the database object
name	VARCHAR(128)	Name of the database object
tabid	INTEGER	Identifying code of table on which the object is defined
state	CHAR(1)	The current state (object mode) of the database object. This value can be one of the following codes: • D = Disabled • E = Enabled • F = Filtering with no integrity-violation errors • G = Filtering with integrity-violation error

A composite index on the **objtype**, **name**, **owner**, and **tabid** columns allows only unique combinations of values. A simple index on the **tabid** column allows duplicate values.

SYSOPCLASSES

The **sysopclasses** system catalog table contains information about operator classes associated with secondary access methods. It contains one row for each operator class that has been defined in the database. The **sysopclasses** table has the following columns.

Column	Type	Explanation
opclassname	VARCHAR(128)	Name of the operator class
owner	VARCHAR(32)	Name of the owner of the operator class
amid	INTEGER	Identifying code of the secondary access method associated with this operator class
opclassid	SERIAL	Identifying code of the operator class
ops	LVARCHAR(2048)	List of names of the operators that belong to this operator class
support	LVARCHAR(2048)	List of names of support functions defined for this operator class

The **opclassid** value corresponds to the **sysams.am_defopclass** value that specifies the default operator class for the secondary access method that the **amid** column specifies.

The **sysopclasses** table has a composite index on the **opclassname** and **owner** columns and an index on **opclassid** column. Both indexes allow only unique values.

SYSPROCAUTH

The **sysprocauth** system catalog table describes the privileges granted on a procedure or function. It contains one row for each set of privileges that is granted. The **sysprocauth** table has the following columns.

Table 1-23. SYSPROCAUTH table column descriptions

Column	Type	Explanation
grantor	VARCHAR(32)	Name of grantor of privileges to access the routine
grantee	VARCHAR(32)	Name of grantee of privileges to access the routine
procid	INTEGER	Unique identifying code of the routine
procauth	CHAR(1)	Type of privilege granted on the routine: e = Execute privilege on routine E = Execute privilege WITH GRANT OPTION

A composite index on the **procid**, **grantor**, and **grantee** columns allows only unique values. A composite index on the **procid** and **grantee** columns allows duplicate values.

SYSPROCBODY

The **sysprocbody** system catalog table describes the compiled version of each procedure or function in the database. Because the **sysprocbody** table stores the text of the routine, each routine can have multiple rows. The **sysprocbody** table has the following columns.

Table 1-24. SYSPROCBODY table column descriptions

Column	Type	Explanation
procid	INTEGER	Unique identifying code for the routine
datakey	CHAR(1)	Type of information in the **data** column: A = Routine alter SQL (will not change this value after update statistics) D = Routine user documentation text E = Time of creation information L = Literal value (that is, literal number or quoted string) P = Interpreter instruction code (p-code) R = Routine return value type list S = Routine symbol table T = Routine text creation SQL

Table 1-24. SYSPROCBODY table column descriptions (continued)

Column	Type	Explanation
seqno	INTEGER	Line number within the routine
data	CHAR(256)	Actual text of the routine

The A flag indicates the procedure modifiers are altered. ALTER ROUTINE statement updates only modifiers and not the routine body. UPDATE STATISTICS updates the query plan and not the routine modifiers, and the value of datakey will not be changed from A. The A flag marks all the procedures and functions that have altered modifiers, including overloaded procedures and functions. The T flag is used for routine creation text.

The **data** column contains actual data, which can be in one of these formats:
- Encoded return values list
- Encoded symbol table
- Literal data
- P-code for the routine
- Compiled code for the routine
- Text of the routine and its documentation

A composite index on the **procid**, **datakey**, and **seqno** columns allows only unique values.

SYSPROCCOLUMNS

The **sysproccolumns** system catalog table stores information about return types and parameter names of all UDRs in SYSPROCEDURES.

A composite index on the **procid** and **paramid** columns in this table allows only unique values.

Table 1-25. SYSPROCCOLUMNS table column descriptions

Column	Type	Explanation
procid	INTEGER	Unique identifying code of the routine
paramid	INTEGER	Unique identifying code of the parameter
paramname	VARCHAR (IDENTSIZE)	Name of the parameter
paramtype	SMALLINT	Identifies the type of parameter
paramlen	SMALLINT	Specifies the length of the parameter
paramxid	INTEGER	Specifies the extended type ID for the parameter
paramattr	INTEGER	0 = Parameter is of unknown type 1 = Parameter is INPUT mode 2 = Parameter is INOUT mode 3 = Parameter is multiple return value 4 = Parameter is OUT mode 5 = Parameter is a return value

SYSPROCEDURES

The **sysprocedures** system catalog table lists the characteristics for each function and procedure that is registered in the database. It contains one row for each routine.

Each function in **sysprocedures** has a unique value, **procid**, called a *routine identifier*. Throughout the system catalog, a function is identified by its routine identifier, not by its name.

The **sysprocedures** table has the following columns.

Table 1-26. SYSPROCEDURES table column descriptions

Column	Type	Explanation
procname	VARCHAR(128)	Name of routine
owner	VARCHAR(32)	Name of owner
procid	SERIAL	Unique identifying code for the routine
mode	CHAR(1)	Mode type: D *or* d = DBA O *or* o = Owner P *or* p = Protected R *or* r = Restricted T *or* t = Trigger
retsize	INTEGER	Compiled size (in bytes) of returned values
symsize	INTEGER	Compiled size (in bytes) of symbol table
datasize	INTEGER	Compiled size (in bytes) of constant data
codesize	INTEGER	Compiled size (in bytes) of routine code
numargs	INTEGER	Number of arguments to routine

isproc	CHAR(1)	Specifies if the routine is a procedure or a function:
		t = procedure
		f = function

specificname	VARCHAR(128)	Specific name for the routine
externalname	VARCHAR(255)	Location of the external routine. This item is language-specific in content and format.
paramstyle	CHAR(1)	Parameter style: I = IBM Informix
langid	INTEGER	Language code (in **sysroutinelangs** table)
paramtypes	RTNPARAMTYPES	Information describing the parameters of the routine

variant	BOOLEAN	Whether the routine is VARIANT or not:
		t = is VARIANT
		f = is not VARIANT

client	BOOLEAN	Reserved for future use

handlesnulls	BOOLEAN	NULL handling indicator:
		t = handles NULLs
		f = does not handle NULLs

percallcost	INTEGER	Amount of CPU per call Integer cost to execute UDR: cost/call - 0 -(2^31-1)
commutator	VARCHAR(128)	Name of commutator function
negator	VARCHAR(128)	Name of the negator function

Table 1-26. SYSPROCEDURES table column descriptions (continued)

Column	Type	Explanation
selfunc	VARCHAR(128)	Name of function to estimate selectivity of the UDR
internal	BOOLEAN	Specifies if the routine can be called from SQL: t = routine is internal, not callable from SQL f = routine is external, callable from SQL
class	CHAR(18)	CPU class by which the routine should be executed
stack	INTEGER	Stack size in bytes required per invocation

parallelizable	BOOLEAN	Parallelization indicator for UDR:
		t = parallelizable
		f = not parallelizable

costfunc	VARCHAR(128)	Name of the cost function for the UDR
selconst	SMALLFLOAT	Selectivity constant for UDR
procflags	INTEGER	For internal use only
collation	CHAR(32)	Collating order at the time when the routine was created

In the **mode** column, the R mode is a special case of the O mode. A routine is in restricted (R) mode if it was created with a specified owner who is different from the routine creator. If routine statements involving a remote database are executed, the database server uses the access privileges of the user who executes the routine instead of the privileges of the routine owner. In all other scenarios, R-mode routines behave the same as O-mode routines.

The database server can create protected routines for internal use. The **sysprocedures** table identifies these protected routines with the letter `P` or `p` in the **mode** column, where `p` indicates an SPL routine. Protected routines have the following restrictions:

- You cannot use the ALTER FUNCTION, ALTER PROCEDURE, or ALTER ROUTINE statements to modify protected routines.
- You cannot use the DROP FUNCTION, DROP PROCEDURE, or DROP ROUTINE statements to unregister protected routines.
- You cannot use the **dbschema** utility to display protected routines.

In earlier versions, protected SPL routines were indicated by a lowercase `p`. Starting with version 9.0, protected SPL routines are treated as DBA routines and cannot be Owner routines. Thus `D` and `O` indicate DBA routines and Owner routines, while `d` and `o` indicate protected DBA routines and protected Owner routines.

The trigger mode designates user-defined SPL routines that can be invoked only from the FOR EACH ROW section of a triggered action.

Important: After you issue the SET SESSION AUTHORIZATION statement, the database server assigns a restricted mode to all Owner routines that you created while using the new identity.

A unique index is defined on the **procid** column. A composite index on the **procname**, **isproc**, **numargs**, and **owner** columns allows duplicate values, as does a composite index on the **specificname** and **owner** columns.

SYSPROCPLAN

The **sysprocplan** system catalog table describes the query-execution plans and dependency lists for data-manipulation statements within each routine. Because different parts of a routine plan can be created on different dates, this table can contain multiple rows for each routine.

Table 1-27. SYSPROCPLAN table column descriptions

Column	Type	Explanation
procid	INTEGER	Identifying code for the routine
planid	INTEGER	Identifying code for the plan
datakey	CHAR(1)	Type of information stored in **data** column: D = Dependency list I = Information record Q = Execution plan
seqno	INTEGER	Line number within the plan
created	DATE	Date when plan was created
datasize	INTEGER	Size (in bytes) of the list or plan
data	CHAR(256)	Encoded (compiled) list or plan

Before a routine is run, its dependency list in the **data** column is examined. If the major version number of a table accessed by the plan has changed, or if any object that the routine uses has been modified since the plan was optimized (for example, if an index has been dropped), then the plan is optimized again. When **datakey** is I, the **data** column stores information about UPDATE STATISTICS and PDQPRIORITY.

It is possible to delete all the plans for a given routine by using the DELETE statement on **sysprocplan**. When the routine is subsequently executed, new plans are automatically generated and recorded in **sysprocplan**. The UPDATE STATISTICS FOR PROCEDURE statement also updates this table.

A composite index on the **procid**, **planid**, **datakey**, and **seqno** columns allows only unique values.

SYSREFERENCES

The **sysreferences** system catalog table lists all referential constraints on columns. It contains a row for each referential constraint in the database.

Table 1-28. SYSREFERENCES table column descriptions

Column	Type	Explanation
constrid	INTEGER	Code uniquely identifying the constraint
primary	INTEGER	Identifying code of the corresponding primary key
ptabid	INTEGER	Identifying code of the table that is the primary key
updrule	CHAR(1)	Reserved for future use; displays an R

Table 1-28. SYSREFERENCES table column descriptions (continued)

Column	Type	Explanation
delrule	CHAR(1)	Whether constraint uses cascading delete or restrict rule: C = Cascading delete R = Restrict (default)
matchtype	CHAR(1)	Reserved for future use; displays an N
pendant	CHAR(1)	Reserved for future use; displays an N

The **constrid** column is indexed and allows only unique values. The **primary** column is indexed and allows duplicate values.

SYSROLEAUTH

The **sysroleauth** system catalog table describes the roles that are granted to users. It contains one row for each role that is granted to a user in the database. The **sysroleauth** table has the following columns.

Table 1-29. SYSROLEAUTH table column descriptions

Column	Type	Explanation
rolename	VARCHAR(32)	Name of the role
grantee	VARCHAR(32)	Name of the grantee of the role
is_grantable	CHAR(1)	Specifies whether the role is grantable: Y = Grantable N = Not grantable

The **is_grantable** column indicates whether the role was granted with the WITH GRANT OPTION of the GRANT statement.

A composite index on the **rolename** and **grantee** columns allows only unique values.

SYSROUTINELANGS

The **sysroutinelangs** system catalog table lists the supported programming languages for user-defined routines (UDRs). It has these columns.

Column	Type	Explanation
langid	SERIAL	Code uniquely identifying a supported language
langname	CHAR(30)	Name of the language, such as C or SPL
langinitfunc	VARCHAR(128)	Name of initialization function for the language
langpath	CHAR(255)	Directory path for the UDR language
langclass	CHAR(18)	Name of the class of the UDR language

An index on the **langname** column allows duplicate values.

SYSSECLABELAUTH

The **sysseclabelauth** system catalog table records the LBAC labels that have been granted to users. It has these columns.

Column	Type	Explanation
GRANTEE	CHAR(32)	The name of the label grantee
secpolicyid	INTEGER	The ID of the security policy to which the security label belongs.
readseclabelid	INTEGER	The security label ID of the security label granted for read access
writeseclabelid	INTEGER	The security label ID of the security label granted for write access

SYSSECLABELCOMPONENTS

The **sysseclabelcomponents** system catalog table records security label components. It has these columns.

Column	Type	Explanation
compname	VARCHAR(128)	Component name
compid	SERIAL	Component ID
comptype	CHAR(1)	The component type: A = array S = set T = tree
numelements	INTEGER	Number of elements in the component
coveringinfo	VARCHAR(128)	Internal encoding information
numalters	SMALLINT	Numbers of alter operations that have been performed on the component

SYSSECLABELCOMPONENTELEMENTS

The **sysseclabelcomponentelements** system catalog table records the values of component elements of security labels. It has these columns.

Column	Type	Explanation
compid	INTEGER	Component ID
element	VARCHAR(32)	Element name
elementencoding	CHAR(8)	Encoded form of the element
parentelement	VARCHAR(32)	The name of the parent elements for tree components. The value is NULL for the following items: Set components Array components Root nodes of a tree component

Column	Type	Explanation
alterversion	SMALLINT	The number of the alter operation when the element is added. This value is used by the **dbexport** and **dbimport** commands.

SYSSECLABELNAMES

The **sysseclabelnames** system catalog table records the security label names. It has these columns.

Column	Type	Explanation
secpolicyid	INTEGER	The ID of the security policy to which the security label belongs.
seclabelname	VARCHAR(128)	The name of the security label
seclabelid	INTEGER	The ID of the security label

SYSSECLABELS

The **sysseclabels** system catalog table records the security label encoding. It has these columns.

Column	Type	Explanation
secpolicyid	INTEGER	ID of the security policy to which the security label belongs
seclabelid	INTEGER	Security label ID
sysseclabelnames	VARCHAR(128)	Security label encoding

SYSSECPOLICIES

The **syssecpolicies** system catalog table records security policies It has these columns.

Column	Type	Explanation
secpolicyname	VARCHAR(128)	Security policy name
secpolicyid	SERIAL	Security policy ID
numcomps	SMALLINT	Number of security label components in the security policy

comptypelist	CHAR(16)	An ordered list of the type of each component in the policy.
		A = array
		S = set
		T = tree
		– = Beyond NUMCOMPS

Column	Type	Explanation
overrideseclabel	CHAR(1)	Indicates the behavior when a user's security label and exemption credentials do not allow them to insert or update a data row with the security that is label provided on the INSERT or UPDATE SQL statement. • Y: The security label provided is ignored and replaced by the user's security label for write access. • N: Return an error when not authorized to write a security label.

SYSSECPOLICYCOMPONENTS

The **syssecpolicycomponents** system catalog table records the components for each security policies. It has these columns.

Column	Type	Explanation
secpolicyid	INTEGER	Security policy ID
compid	INTEGER	ID of a component of the label security policy
compno	SMALLINT	Position of the security label component as it exists in the security policy, starting with position 1.

SYSSECPOLICYEXEMPTIONS

The **syssecpolicyexemptions** system catalog table records the exemptions that have been given to users. It has these columns.

Column	Type	Explanation
grantee	CHAR(32)	The user who has this exemption
secpolicyid	INTEGER	ID of the policy on which the exemption is granted

Column	Type	Explanation
exemption	CHAR(6)	The exemption given to the user who is identified in the GRANTEE column. The six characters have the following meanings: 1 = Read array 2 = Read set 3 = Read tree 4 = Write array 5 = Write set 6 = Write tree Each character has one of the following values: E = Exempt D = Write down exemption U = Write up exemption – = No exemption

SYSSEQUENCES

The **syssequences** system catalog table lists the sequence objects that exist in the database. The **syssequences** table has the following columns.

Column	Type	Explanation
seqid	SERIAL	Code uniquely identifying the sequence object
tabid	INTEGER	Identifying code of the sequence as a table object
start_val	INT8	Starting value of the sequence
inc_val	INT8	Value of the increment between successive values
max_val	INT8	Largest possible value of the sequence
min_val	INT8	Smallest possible value of the sequence
cycle	CHAR(1)	Zero means NOCYCLE, 1 means CYCLE
restart_val	INT8	Starting value of the sequence after ALTER SEQUENCE RESTART was run
cache	INTEGER	Number of preallocated values in sequence cache
order	CHAR(1)	Zero means NOORDER, 1 means ORDER

SYSSURROGATEAUTH

The **syssurrogateauth** system catalog table stores trusted user and surrogate user information.

The **syssurrogateauth** system catalog table is populated when the GRANT SETSESSIONAUTH statement is run. Users or roles specified in the TO clause are added to **trusteduser** column. Users specified in the ON clause are added to **surrogateuser** column.

For example, consider the following statement:

```
GRANT SETSESSIONAUTH ON bill, john TO mary, peter;
```

Entries in the **syssurrogateauth** table are created as follows:

```
trusteduser    surrogateuser

mary           bill
mary           john
peter          bill
peter          john
```

The **syssurrogateauth** table has the following columns.

Table 1-30. SYSSURROGATEAUTH table column descriptions

Column	Type	Explanation
trusteduser	CHAR(32)	Trusted user name or role.
surrogateuser	CHAR(32)	Surrogate user name.

SYSSYNONYMS

The **syssynonyms** system catalog table is unused. The **syssyntable** table describes synonyms. The **syssynonyms** system catalog table has the following columns.

Table 1-31. SYSSYNONYMS table column descriptions

Column	Type	Explanation
owner	VARCHAR(32)	Name of the owner of the synonym
synname	VARCHAR(128)	Name of the synonym
created	DATE	Date when the synonym was created
tabid	INTEGER	Identifying code of a table, sequence, or view

A composite index on the **owner** and **synonym** columns allows only unique values. The **tabid** column is indexed and allows duplicate values.

SYSSYNTABLE

The **syssyntable** system catalog table outlines the mapping between each public or private synonym and the database object (table, sequence, or view) that it represents. It contains one row for each entry in the **systables** table that has a **tabtype** value of `P`or `S`. The **syssyntable** table has the following columns.

Column	Type	Explanation
tabid	INTEGER	Identifying code of the public synonym
servername	VARCHAR(128)	Name of an external database server
dbname	VARCHAR(128)	Name of an external database
owner	VARCHAR(32)	Name of the owner of an external object
tabname	VARCHAR(128)	Name of an external table or view
btabid	INTEGER	Identifying code of a base table, sequence, or view

ANSI-compliant databases do not support public synonyms; their **syssyntable** tables can describe only synonyms whose **syssyntable.tabtype** value is `P`.

If you define a synonym for an object that is in your current database, only the **tabid** and **btabid** columns are used. If you define a synonym for a table that is external to your current database, the **btabid** column is not used, but the **tabid**, **servername**, **dbname**, **owner**, and **tabname** columns are used.

The **tabid** column maps to **systables.tabid**. With the **tabid** information, you can determine additional facts about the synonym from **systables**.

An index on the **tabid** column allows only unique values. The **btabid** column is indexed to allow duplicate values.

SYSTABAMDATA

The **systabamdata** system catalog table stores the table-specific hashing parameters of tables that were created with a primary access method.

The **systabamdata** table has the following columns.

Table 1-32. SYSTABAMDATA table column descriptions

Column	Type	Explanation
tabid	INTEGER	Identifying code of the table
am_param	LVARCHAR(8192)	Access method parameter choices
am_space	VARCHAR(128)	Name of the storage space holding the data values

The **am_param** column stores configuration parameters that determine how a primary access method accesses a given table. Each configuration parameter in the **am_param** list has the format *keyword=value* or *keyword*.

The **am_space** column specifies the location of the table. It might be located in a cooked file, a different database, or an sbspace within the database server.

The **tabid** column is the primary key to the **systables** table. This column is indexed and must contain unique values.

SYSTABAUTH

The **systabauth** system catalog table describes each set of privileges that are granted on a table, view, sequence, or synonym. It contains one row for each set of table privileges that are granted in the database; the REVOKE statement can modify a row. The **systabauth** table has the following columns.

Table 1-33. SYSTABAUTH table column descriptions

Column	Type	Explanation
grantor	VARCHAR(32)	Name of the grantor of privilege
grantee	VARCHAR(32)	Name of the grantee of privilege
tabid	INTEGER	Value from **systables.tabid** for database object

Table 1-33. SYSTABAUTH table column descriptions (continued)

Column	Type	Explanation
tabauth	CHAR(9) CHAR(8)	Pattern that specifies privileges on the table, view, synonym, or sequence: s *or* S = Select; u *or* U = Update; * = Column-level privilege; i *or* I = Insert; d *or* D = Delete; x *or* X = Index; a *or* A = Alter; r *or* R = References; n *or* N = Under privilege

If the **tabauth** column shows a privilege code in uppercase (for example, S for Select), this indicates that the user also has the option to grant that privilege to others. Privilege codes listed in lowercase (for example, s for select) indicate that the user has the specified privilege, but cannot grant it to others.

A hyphen (-) indicates the absence of the privilege corresponding to that position within the **tabauth** pattern.

A **tabauth** value with an asterisk (*) means column-level privileges exist; see also **syscolauth** (page "SYSINDEXES" on page 1-33). (In DB-Access, the **Privileges** option of the **Info** command for a specified table can display the column-level privileges on that table.)

A composite index on **tabid**, **grantor**, and **grantee** allows only unique values. A composite index on **tabid** and **grantee** allows duplicate values.

SYSTABLES

The **systables** system catalog table contains a row for each table object (a table, view, synonym, or in IBM Informix, a sequence) that has been defined in the database, including the tables and views of the system catalog.

Table 1-34. SYSTABLES table column descriptions

Column	Type	Explanation
tabname	VARCHAR(128)	Name of table, view, synonym, or sequence
owner	CHAR(32)	Owner of table (user **informix** for system catalog tables and *username* for database tables)
partnum	INTEGER	Physical storage location code
tabid	SERIAL	System-assigned sequential identifying number
rowsize	SMALLINT	Maximum row size in bytes (< 32,768)
ncols	SMALLINT	Number of columns in the table
nindexes	SMALLINT	Number of indexes on the table
nrows	FLOAT	Number of rows in the table

Table 1-34. SYSTABLES table column descriptions (continued)

Column	Type	Explanation
created	DATE	Date when table was created or last modified
version	INTEGER	Number that changes when table is altered
tabtype	CHAR(1)	Code indicating the type of table object: • T = Table • E = External Table • V = View • Q = Sequence • P = Private synonym • S = Public synonym (Type S is unavailable in an ANSI-compliant database.)
locklevel	CHAR(1)	Lock mode for the table: • B = Page and row level • P = Page level • R = Row level
npused	FLOAT	Number of data pages that have ever been initialized in the tablespace by the database server
fextsize	INTEGER	Size of initial extent (in KB)
nextsize	INTEGER	Size of all subsequent extents (in KB)
flags	SMALLINT	Codes for classifying permanent tables: **ROWID** 1 - Has rowid column defined **UNDER** 2 - Table created under a supertable **VIEWREMOTE** 4 - View is based on a remote table **CDR** 8 - Has CDRCOLS defined **RAW** 16 - (Informix) RAW table **EXTERNAL** 32- External table **AUDIT** 64 - Audit table attribute - FGA **AQT** 128 - View is an AQT for DWA offloading **VIRTAQT** 256 - View is a virtual AQT
site	VARCHAR(128)	Reserved for future use
dbname	VARCHAR(128)	Reserved for future use
type_xid	INTEGER	Code from **sysxtdtypes.extended_id** for typed tables, or `0` for untyped tables

Table 1-34. SYSTABLES table column descriptions (continued)

Column	Type	Explanation
am_id	INTEGER	Access method code (key to **sysams** table) NULL or 0 indicates built-in storage manager
pagesize	INTEGER	The pagesize, in bytes, of the dbspace (or dbspaces, if the table is fragmented) where the table data resides.
ustlowts	DATETIME YEAR TO FRACTION (5)	When table, row, and page-count statistics were last recorded
secpolicyid	INTEGER	ID of the SECURITY policy attached to the table. NULL for non-protected tables
protgranularity	CHAR(1)	LBAC granularity level: • R: Row level granularity • C: Column level granularity • B: Both column and row granularity • Blank for non-protected tables
statlevel	CHAR(1)	Statistics level • T = table • F = fragment • A = automatic
statchange	SMALLINT	For internal use only

Each table, view, sequence, and synonym recorded in the **systables** table is assigned a **tabid**, which is a system-assigned SERIAL value that uniquely identifies the object. The first 99 **tabid** values are reserved for the system catalog. The **tabid** of the first user-defined table object in a database is always 100.

The **tabid** column is indexed and contains only unique values. A composite index on the **tabname** and **owner** columns also requires unique values.

The version column contains an encoded number that is stored in **systables** when a new table is created. Portions of this value are incremented when data-definition statements, such as ALTER INDEX, ALTER TABLE, DROP INDEX, and CREATE INDEX, are performed on the table.

In the **flags** column, ST_RAW represents a nonlogging permanent table in a database that supports transaction logging.

The setting of the SQL_LOGICAL_CHAR parameter is encoded into the **systables.flags** column value in the row that describes the ' **VERSION**' table. Note the leading blank space in the identifier of this system-generated table.

To determine whether the database enables the SQL_LOGICAL_CHAR configuration parameter, which can apply logical character semantics to the declarations of character columns, you can execute the following query:

```
SELECT flags INTO $value FROM 'informix'.systables WHERE tabname = ' VERSION';
```

Because the SQL_LOGICAL_CHAR setting is encoded in the two least significant bits of the " **VERSION.flags**" value, you can calculate its setting from the returned **flags** value by the following formula:

```
SQL_LOGICAL_CHAR = (value & 0x03) + 1
```

Here `&` is the bitwise `AND` operator. Any SQL_LOGICAL_CHAR setting greater than 1 indicates that SQL_LOGICAL_CHAR was enabled when the database was created, and that explicit or default maximum size specifications of character columns are multiplied by that setting.

When a prepared statement that references a database table is executed, the version value is checked to make sure that nothing has changed since the statement was prepared. If the version value has been changed by DDL operations that modified the table schema while automatic recompilation was disabled by the IFX_AUTO_REPREPARE setting of the SET ENVIRONMENT statement, the prepared statement is not executed, and you must prepare the statement again.

The **npused** column does not reflect the number of pages used for BYTE or TEXT data, nor the number of pages that are freed in DELETE or TRUNCATE operations.

The **nrows** column and the **npused** columns might not accurately reflect the number of rows and the number of data pages used by an external table unless the NUMROWS clause was specified when the external table was created. See the *IBM Informix Administrator's Guide* for more information.

The **systables** table has two rows that store information about the database locale: GL_COLLATE with a **tabid** of `90` and GL_CTYPE with a **tabid** of `91`. To view these rows, enter the following SELECT statement:

```
SELECT * FROM systables WHERE tabid=90 OR tabid=91;
```

SYSTRACECLASSES

The **systraceclasses** system catalog table contains the names and identifiers of trace classes. The **systraceclasses** table has the following columns.

Table 1-35. SYSTRACECLASSES table column descriptions

Column	Type	Explanation
name	CHAR(18)	Name of the class of trace messages
classid	SERIAL	Identifying code of the trace class

A *trace class* is a category of trace messages that you can use in the development and testing of new DataBlade modules and user-defined routines. Developers use the tracing facility by calling the appropriate DataBlade API routines within their code.

To create a new trace class, insert a row directly into the **systraceclasses** table. By default, all users can view this table, but only users with the DBA privilege can modify it.

The database cannot support tracing unless the MITRACE_OFF configuration parameter is undefined.

A unique index on the **name** column requires each trace class to have a unique name. The database server assigns to each class a unique sequential code. The index on this **classid** column also allows only unique values.

SYSTRACEMSGS

The **systracemsgs** system catalog table stores internationalized trace messages that you can use in debugging user-defined routines.

The **systracemsgs** table has the following columns.

Table 1-36. SYSTRACEMSGS table column descriptions

Column	Type	Explanation
name	VARCHAR(128)	Name of the message
msgid	SERIAL	Identifying code of the message template
locale	CHAR(36)	Locale with which this version of the message is associated (for example, **en_us.8859-1**)
seqno	SMALLINT	Reserved for future use
message	VARCHAR(255)	The message text

DataBlade module developers create a trace message by inserting a row directly into the **systracemsgs** table. After a message is created, the development team can specify it either by name or by **msgid** code, using trace statements that the DataBlade API provides.

To create a trace message, you must specify its name, locale, and text. By default, all users can view the **systracemsgs** table, but only users with the DBA privilege can modify it.

The database cannot support tracing unless the MITRACE_OFF configuration parameter is undefined.

A unique composite index is defined on the **name** and **locale** columns. Another unique index is defined on the **msgid** column.

SYSTRIGBODY

The **systrigbody** system catalog table contains the ASCII text of the trigger definition and the linearized code for the trigger. *Linearized code* is binary data and code that is represented in ASCII format.

Important: The database server uses the linearized code that is stored in **systrigbody**. You must not alter the content of rows that contain linearized code.

The **systrigbody** table has the following columns.

Table 1-37. SYSTRIGBODY table column descriptions

Column	Type	Explanation
trigid	INTEGER	Identifying code of the trigger

Table 1-37. SYSTRIGBODY table column descriptions (continued)

Column	Type	Explanation
datakey	CHAR(1)	Code specifying the type of data: A = ASCII text for the body, triggered actions B = Linearized code for the body D = English text for the header, trigger definition H = Linearized code for the header S = Linearized code for the symbol table
seqno	INTEGER	Page number of this data segment
data	CHAR(256)	English text or linearized code

A composite index on the **trigid**, **datakey**, and **seqno** columns allows only unique values.

SYSTRIGGERS

The **systriggers** system catalog table contains information about the SQL triggers in the database. This information includes the triggering event and the correlated reference specification for the trigger. The **systriggers** table has the following columns.

Table 1-38. SYSTRIGGERS table column descriptions

Column	Type	Explanation
trigid	SERIAL	Identifying code of the trigger
trigname	VARCHAR(128)	Name of the trigger
owner	VARCHAR(32)	Name of the owner of the trigger
tabid	INTEGER	Identifying code of the triggering table

event	CHAR(1)	Code for the type of triggering event:
		D = Delete trigger
		I = Insert trigger
		U = Update trigger
		S = Select trigger
		d = INSTEAD OF Delete trigger
		i = INSTEAD OF Insert trigger
		u = INSTEAD OF Update trigger

old	VARCHAR(128)	Name of value before update
new	VARCHAR(128)	Name of value after update
mode	CHAR(1)	Reserved for future use
collation	CHAR(32)	Collating order at the time when the routine was created

A composite index on the **trigname** and **owner** columns allows only unique values. An index on the **trigid** column also requires unique values. An index on the **tabid** column allows duplicate values.

SYSUSERS

The **sysusers** system catalog table lists the authorization identifier of every individual user, or public for the PUBLIC group, who holds database-level access privileges. This table also lists the name of every role that holds access privileges on any object in the database.

This system catalog table has the following columns:

Table 1-39. SYSUSERS table column descriptions

Column	Type	Explanation
username	VARCHAR(32)	Name of the database user or role. An index on **username** allows only unique values. The **username** value can be the login name of a user or the name of a role.
usertype	CHAR(1)	Code specifying the highest database-level privilege held by **username**, where **username** is an individual user or the PUBLIC group, or a role name. The valid codes are: D = DBA (all privileges) R = Resource (create UDRs, UDTs, permanent tables, and indexes) C = Connect (work with existing tables) G = Role U = Default role. When a user is assigned a default role, an implicit connection to the database is granted to the user. This is the role the user has before being granted a C, D, or R role.
priority	SMALLINT	Reserved for future use.
password	CHAR(16)	Reserved for future use.
defrole	VARCHAR(32)	Name of the default role.

SYSVIEWS

The **sysviews** system catalog table describes each view in the database. Because it stores the SELECT statement that created the view, **sysviews** can contain multiple rows for each view. It has the following columns.

Column	Type	Explanation
tabid	INTEGER	Identifying code of the view
seqno	SMALLINT	Line number of the SELECT statement

Column	Type	Explanation
viewtext	CHAR(256)	Actual SELECT statement used to create the view

A composite index on **tabid** and **seqno** allows only unique values.

SYSVIOLATIONS

The **sysviolations** system catalog table stores information about constraint violations for base tables.

This table is updated when the DELETE, INSERT, MERGE, or UPDATE statement detects a violation of an enabled constraint or unique index in a database table for which the START VIOLATIONS TABLE statement of SQL has created an associated violations table (and for Informix, a diagnostics table). For each base table that has an active violations table, the **sysviolations** table has a corresponding row, with the following columns.

Column	Type	Explanation
targettid	INTEGER	Identifying code of the *target table* (the base table on which the violations table and the diagnostic table are defined)
viotid	INTEGER	Identifying code of the violations table
diatid	INTEGER	Identifying code of the diagnostics table
maxrows	INTEGER	Maximum number of rows that can be inserted into the diagnostics table by a single insert, update, or delete operation on a target table that has a filtering mode object defined on it.

The **maxrows** column also signifies the maximum number of rows that can be inserted in the diagnostics table during a single operation that enables a disabled object or that sets a disabled object to filtering mode (provided that a diagnostics table exists for the target table). If no maximum is specified for the diagnostics or violations table, then **maxrows** contains a NULL value.

The primary key of this table is the **targettid** column. An additional unique index is also defined on the **viotid** column.

IBM Informix also has a unique index on the **diatid** column.

SYSXADATASOURCES

The **sysxadatasources** system catalog table stores XA data sources.

The **sysxadatasources** table has the following columns.

Column	Type	Explanation
xa_datasrc_owner	CHAR(32)	The user ID of the XA data source owner
xa_datasrc_name	VARCHAR(128)	The name of the XA data source
xa_datasrc_rmid	SERIAL	Unique RMID of the XA data source
xa_source_typeid	INTEGER	XA data source type ID

SYSXASOURCETYPES

The **sysxasourcetypes** system catalog table stores XA data source types.

The **sysxasourcetypes** table has the following columns.

Column	Type	Explanation
xa_source_typeid	SERIAL	A unique identifier for the source type
xa_source_owner	CHAR(32)	The user ID of the owner
xa_source_name	VARCHAR(128)	The name of the source type
xa_flags	INTEGER	
xa_version	INTEGER	
xa_open	INTEGER	UDR ID of xa_open_entry
xa_close	INTEGER	UDR ID of xa_close_entry
xa_start	INTEGER	UDR ID of xa_start entry
xa_end	INTEGER	UDR ID of xa_end_entry
xa_rollback	INTEGER	UDR ID of xa_rollback_entry
xa_prepare	INTEGER	UDR ID of xa_prepare_entry
xa_commit	INTEGER	UDR ID of xa_commit_entry
xa_recover	INTEGER	UDR ID of xa_recover_entry
xa_forget	INTEGER	UDR ID of xa_forget_entry
xa_complete	INTEGER	UDR ID of xa_complete_entry

SYSXTDDESC

The **sysxtddesc** system catalog table provides a text description of each user-defined data type (UDT) defined in the database. The **sysxtddesc** table has the following columns.

Column	Type	Explanation
extended_id	INTEGER	Code uniquely identifying the extended data types
seqno	SMALLINT	Value to order and identify one line of the description of the UDT A new line is created only if the remaining text string is larger than 255 bytes.
description	CHAR(256)	Textual description of the extended data type

A composite index on **extended_id** and **seqno** allows duplicate values.

SYSXTDTYPEAUTH

The **sysxtdtypeauth** system catalog table identifies the privileges on each UDT (user-defined data type).

The **sysxtdtypeauth** table contains one row for each set of privileges granted and has the following columns:

Column	Type	Explanation
grantor	VARCHAR(32)	Name of grantor of privilege
grantee	VARCHAR(32)	Name of grantee of privilege
type	INTEGER	Code identifying the UDT
auth	CHAR(2)	Code identifying privileges on the UDT: n *or* N = Under privilege u *or* U = Usage privilege

If the privilege code in the **auth** column is upper case (for example, 'U' for usage), a user who has this privilege can also grant it to others. If the code is in lower case, a user who has the privilege cannot grant it to others.

A composite index on **type**, **grantor**, and **grantee** allows only unique values. A composite index on the **type** and **grantee** columns allows duplicate values.

SYSXTDTYPES

The **sysxtdtypes** system catalog table has an entry for each UDT (user-defined data type), including opaque and distinct data types and complex data types (named ROW types, unnamed ROW types, and COLLECTION types), that is defined in the database.

The **sysxtdtypes** table has the following columns.

Table 1-40. SYSXTDTYPES table column descriptions

Column	Type	Explanation
extended_id	SERIAL	Unique identifying code for extended data type
domain	CHAR(1)	Code for the domain of the UDT
mode	CHAR(1)	Code classifying the UDT: • B = Base (opaque) type • C = Collection type or unnamed ROW type • D = Distinct type • R = Named ROW type • S = Reserved for internal use • T = Reserved for internal use • ' ' (blank) = Built-in type
owner	VARCHAR(32)	Name of the owner of the UDT
name	VARCHAR(128)	Name of the UDT
type	SMALLINT	Code classifying the UDT

Table 1-40. SYSXTDTYPES table column descriptions (continued)

Column	Type	Explanation
source	INTEGER	The **sysxtdtypes** reference (for distinct types only) Zero (0) indicates that a distinct UDT was created from a built-in data type.
maxlen	INTEGER	The maximum length for variable-length data types Zero indicates a fixed-length UDT.
length	INTEGER	The length in bytes for fixed-length data types Zero indicates a variable-length UDT.
byvalue	CHAR(1)	'T' = UDT is passed by value 'F' = UDT is not passed by value
cannothash	CHAR(1)	'T' = UDT is hashable by default hash function 'F' = UDT is not hashable by default function
align	SMALLINT	Alignment (= 1, 2, 4, *or* 8) for this UDT
locator	INTEGER	Locator key for unnamed ROW type

Each extended data type is characterized by a unique identifier, called an extended identifier (**extended_id**), a data type identifier (**type**), and the length and description of the data type.

For distinct types created from built-in data types, the **type** column codes correspond to the value of the **syscolumns.coltype** column (indicating the source type) as listed on page "SYSCOLUMNS" on page 1-17, but incremented by the hexadecimal value 0x0000800. The file `$INFORMIXDIR/incl/esql/sqltypes.h` contains information about **sysxtdtypes.type** and **syscolumns.coltype** codes.

An index on the **extended_id** column allows only unique values. An index on the **locator** column allows duplicate values, as does a composite index on the **name** and **owner** columns. A composite index on the **type** and **source** columns also allows duplicate values.

Information Schema

The Information Schema consists of read-only views that provide information about all the tables, views, and columns in the current database server to which you have access. These views also provide information about SQL dialects (such as IBM Informix, Oracle, or Sybase) and SQL standards. Note that unlike a system catalog, whose tables describes an individual database, these views describe the IBM Informix instance, rather than a single database.

This version of the Information Schema views is an X/Open CAE standard. These standards are provided so that applications developed on other database systems can obtain IBM Informix system catalog information without accessing the IBM Informix system catalog tables directly.

Important: Because the X/Open CAE standard for Information Schema views differs from ANSI-compliant Information Schema views, it is recommended that you do not install the X/Open CAE Information Schema views on ANSI-compliant databases.

The following Information Schema views are available:
- **tables**
- **columns**
- **sql_languages**
- **server_info**

Sections that follow contain information about how to generate and access Information Schema views and information about their structure.

Generating the Information Schema Views

The Information Schema views are generated automatically when you, as DBA, run the following DB-Access command:

```
dbaccess database-name $INFORMIXDIR/etc/xpg4_is.sql
```

The views display data from the system catalog tables. If tables, views, or routines exist with any of the same names as the Information Schema views, you must either rename those database objects or rename the views in the script before you can install the views. You can drop the views with the DROP VIEW statement on each view. To re-create the views, rerun the script.

Important: In addition to the columns specified for each Information Schema view, individual vendors might include additional columns or change the order of the columns. It is recommended that applications not use the forms SELECT * or SELECT table-name* to access an Information Schema view.

Accessing the Information Schema Views

All Information Schema views have the Select privilege granted to PUBLIC WITH GRANT OPTION so that all users can query the views. Because no other privileges are granted on the Information Schema views, they cannot be updated.

You can query the Information Schema views as you would query any other table or view in the database.

Structure of the Information Schema Views

The following Information Schema views are described in this section:
- **tables**
- **columns**
- **sql_languages**
- **server_info**

In order to accept long identifier names, most of the columns in the views are defined as VARCHAR data types with large maximum sizes.

The tables Information Schema View

The **tables** Information Schema view contains one row for each table to which you have access. It contains the following columns.

Column	Data Type	Explanation
table_schema	VARCHAR(32)	Name of owner of table
table_name	VARCHAR(128)	Name of table or view
table_type	VARCHAR(128)	BASE TABLE for table or VIEW for view
remarks	VARCHAR(255)	Reserved for future use

The visible rows in the **tables** view depend on your privileges. For example, if you have one or more privileges on a table (such as Insert, Delete, Select, References, Alter, Index, or Update on one or more columns), or if privileges are granted to PUBLIC, you see the row that describes that table.

The columns Information Schema View

The **columns** Information Schema view contains one row for each accessible column. It contains the following columns.

Table 1-41. Description of the columns Information Schema View

Column	Data Type	Explanation
table_schema	VARCHAR(128)	Name of owner of table
table_name	VARCHAR(128)	Name of table or view
column_name	VARCHAR(128)	Name of the column in the table or view
ordinal_position	INTEGER	Position of the column within its table The **ordinal_position** value is a sequential number that starts at 1 for the first column. This is the IBM Informix extension to XPG4.
data_type	VARCHAR(254)	Name of the data type of the column, such as CHARACTER or DECIMAL
char_max_length	INTEGER	Maximum length (in bytes) for character data types; NULL otherwise
numeric_precision	INTEGER	Uses one of the following values: • Total number of digits for exact numeric data types (DECIMAL, INTEGER, MONEY, SMALLINT) • Number of digits of mantissa precision (machine-dependent) for approximate data types (FLOAT, SMALLFLOAT) • NULL for all other data types.
numeric_prec_radix	INTEGER	Uses one of the following values: • 2 = Approximate data types (FLOAT and SMALLFLOAT) • 10 = Exact numeric data types (DECIMAL, INTEGER, MONEY, and SMALLINT) • NULL for all other data types

Table 1-41. Description of the columns Information Schema View (continued)

Column	Data Type	Explanation
numeric_scale	INTEGER	Number of significant digits to the right of the decimal point for DECIMAL and MONEY data types 0 for INTEGER and SMALLINT types NULL for all other data types
datetime_precision	INTEGER	Number of digits in the fractional part of the seconds for DATE and DATETIME columns; NULL otherwise This column is the IBM Informix extension to XPG4.
is_nullable	VARCHAR(3)	Indicates whether a column allows NULL values; either YES or NO
remarks	VARCHAR(254)	Reserved for future use

The sql_languages Information Schema View

The **sql_languages** Information Schema view contains a row for each instance of conformance to standards that the current database server supports. The **sql_languages** view contains the following columns.

Column	Data Type	Explanation
source	VARCHAR(254)	Organization defining this SQL version
source_year	VARCHAR(254)	Year the source document was approved
conformance	VARCHAR(254)	Standard to which the server conforms
integrity	VARCHAR(254)	Indication of whether this is an integrity enhancement feature; either `YES` or `NO`
implementation	VARCHAR(254)	Identification of the SQL product of the vendor
binding_style	VARCHAR(254)	Direct, module, or other binding style
programming_lang	VARCHAR(254)	Host language for which binding style is adapted

The **sql_languages** view is completely visible to all users.

The server_info Information Schema View

The **server_info** Information Schema view describes the database server to which the application is currently connected. It contains two columns.

Column	Data Type	Explanation
server_attribute	VARCHAR(254)	An attribute of the database server
attribute_value	VARCHAR(254)	Value of the **server_attribute** as it applies to the current database server

Each row in this view provides information about one attribute. X/Open-compliant databases must provide applications with certain required information about the database server.

The **server_info** view includes the following **server_attribute** information.

server_attribute	Explanation
identifier_length	Maximum number of bytes for a user-defined identifier
row_length	Maximum number of bytes in a row
userid_length	Maximum number of bytes in a user name
txn_isolation	Initial transaction isolation level for the database server: `Read Uncommitted` (= Default isolation level for databases with no transaction logging; also called `Dirty Read`) `Read Committed` (= Default isolation level for databases that are not ANSI-compliant, but that support explicit transaction logging) `Serializable` (= Default isolation level for ANSI-compliant databases; also called `Repeatable Read`)
collation_seq	Assumed ordering of the character set for the database server The following values are possible: `ISO 8859-1 EBCDIC` The default IBM Informix representation shows ISO 8859-1.

The **server_info** view is completely visible to all users.

Chapter 2. Data types

Every column in a table in a database is assigned a data type. The data type precisely defines the kinds of values that you can store in that column.

These topics describe built-in and extended data types, casting between two data types, and operator precedence.

Summary of data types

IBM Informix supports the most common set of built-in data types. Additionally, an extended set of data types are supported on the database server.

You can use both *built-in* data types (which are system-defined) and *extended* data types (which you can define) in the following ways:
- Use them to create columns within database tables.
- Declare them as arguments and as returned types of routines.
- Use them as base types from which to create DISTINCT data types.
- Cast them to other data types.
- Declare and access host variables of these types in SPL and ESQL/C.

You assign data types to columns with the CREATE TABLE statement and change them with the ALTER TABLE statement. When you change an existing column data type, all data is converted to the new data type, if possible.

For information about the ALTER TABLE and CREATE TABLE statements, on SQL statements that create specific data types, that create and drop casts, and on other data type topics, see the *IBM Informix Guide to SQL: Syntax*.

For information about how to create and use complex data types supported by IBM Informix, see the *IBM Informix Database Design and Implementation Guide*. For information about how to create user-defined data types, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Some data types can be used in distributed SQL operations, while others can be used only in SQL operations within the same database.

Built-in data types supported in local and distributed SQL operations

The following table lists all of the built-in SQL data types that Informix supports. These built-in SQL data types are valid in all Informix SQL transactions, including data-manipulation language (DML) operations of these types:
- Operations on objects in the local database
- Cross-database operations on objects in databases of the local server instance
- Cross-server operations on objects in databases of two or more database server instances

Table 2-1. Data types supported in all operations

Data type	Explanation
"BIGINT data type" on page 2-6	Stores 8-byte integer values from -(2^{63} -1) to 2^{63} -1
"BIGSERIAL data type" on page 2-6	Stores sequential, 8-byte integers from 1 to 2^{63} -1
BSON and JSON built-in opaque data types	The BSON data type is the binary representation of a JSON data type format for serializing JSON documents. The JSON data type is a plain text format for entering and displaying structured data.
"BYTE data type" on page 2-8	Stores any kind of binary data, up to 2^{31} bytes in length
"CHAR(n) data type" on page 2-9	Stores character strings; collation is in code-set order
"CHARACTER(n) data type" on page 2-10	Is a synonym for CHAR
"CHARACTER VARYING(m,r) data type" on page 2-10	Stores character strings of varying length (ANSI-compliant); collation is in code-set order
"DATE data type" on page 2-12	Stores calendar dates
"DATETIME data type" on page 2-12	Stores calendar date combined with time of day
"DEC data type" on page 2-15	Is a synonym for DECIMAL
"DECIMAL" on page 2-15	Stores floating-point numbers with definable precision; if database is ANSI-compliant, the scale is zero
"DECIMAL (p,s) Fixed Point" on page 2-16	Stores fixed-point numbers of defined scale and precision
"DOUBLE PRECISION data types" on page 2-18	Synonym for FLOAT
"FLOAT(n)" on page 2-18	Stores double-precision floating-point numbers corresponding to the **double** data type in C
"INT data type" on page 2-19	Is a synonym for INTEGER
"INT8" on page 2-19	Stores 8-byte integer values from -(2^{63} -1) to 2^{63} -1
"INTEGER data type" on page 2-19	Stores whole numbers from -2,147,483,647 to +2,147,483,647
"INTERVAL data type" on page 2-19	Stores a span of time (or level of effort) in units of *years* and *months*.
"INTERVAL data type" on page 2-19	Stores a span of time in a contiguous set of units of *days*, *hours*, *minutes*, *seconds*, and *fractions of a second*
"MONEY(p,s) data type" on page 2-24	Stores currency amounts
"NCHAR(n) data type" on page 2-25	Same as CHAR, but can support localized collation
"NUMERIC(p,s) data type" on page 2-26	Synonym for DECIMAL(*p*,*s*)
"NVARCHAR(m,r) data type" on page 2-26	Same as VARCHAR, but can support localized collation

Table 2-1. Data types supported in all operations (continued)

Data type	Explanation
“REAL data type” on page 2-27	Is a synonym for SMALLFLOAT
“SERIAL(n) data type” on page 2-29	Stores sequential integers (> 0) in positive range of INT
“SERIAL8(n) data type” on page 2-30	Stores sequential integers (> 0) in positive range of INT8
“SMALLFLOAT” on page 2-33	Stores single-precision floating-point numbers corresponding to the **float** data type of the C language
“SMALLINT data type” on page 2-33	Stores whole numbers from -32,767 to +32,767
“TEXT data type” on page 2-33	Stores any kind of text data, up to 2^{31} bytes in length
“VARCHAR(m,r) data type” on page 2-35	Stores character strings of varying length (up to 255 bytes); collation is in code-set order

In cross-server MERGE operations, the source table (but not the target table) can be in a database of a remote Informix server.

For the character data types (CHAR, CHAR VARYING, LVARCHAR, NCHAR, NVARCHAR, and VARCHAR), a data string can include letters, digits, punctuation, whitespace, diacritical marks, ligatures, and other printable symbols from the code set of the database locale. For **UTF-8** and for code sets of some East Asian locales, multibyte characters are supported within data strings.

Built-in data types supported only in local database SQL operations

The following table lists the data types that Informix supports only for use in SQL operations in a local database.

Table 2-2. Data types supported in a local database

Data type	Explanation
“BLOB data type” on page 2-7	Stores binary data in random-access chunks
The binary18 data type	Stores 18 byte binary-encoded strings
The binaryvar data type	Stores binary-encoded strings with a maximum length of 255 bytes
“BOOLEAN data type” on page 2-8	Stores Boolean values true and false
“CLOB data type” on page 2-11	Stores text data in random-access chunks
“DISTINCT data types” on page 2-17	Stores data in a user-defined type that has the same format as a source type on which it is based, but its casts and functions can differ from those on the source type
Calendar data type	Stores a calendar for a TimeSeries data type
CalendarPattern data type	Stores the structure of the calendar pattern for a Calendar data type

Table 2-2. Data types supported in a local database (continued)

Data type	Explanation
"IDSSECURITYLABEL data type" on page 2-18	Stores LBAC security label objects.
"LIST(e) data type" on page 2-22	Stores a sequentially ordered collection of elements, all of the same data type, *e*; allows duplicate values
The lld_locator data type	Stores a large object identifier
The lld_lob data type	Stores the location of a smart large object and specifies whether the object contains binary or character data
"LVARCHAR(m) data type" on page 2-23	Stores variable-length strings of up to 32,739 bytes
"MULTISET(e) data type" on page 2-25	Stores a non-ordered collection of values, with elements all of the same data type, *e*; allows duplicate values.
The node data type for querying hierarchical data	Stores a combination of integers and decimal points that represents hierarchical relationships, of variable length up to 256 characters
"OPAQUE data types" on page 2-26	Stores a user-defined data type whose internal structure is inaccessible to the database server
"ROW data type, Named" on page 2-27	Stores a named ROW type
"ROW data type, Unnamed" on page 2-28	Stores an unnamed ROW type
"SET(e) data type" on page 2-31	Stores a non-ordered collection of elements, all of the same data type, *e*; does not allow duplicate values
ST_LineString data type	Stores a one-dimensional object as a sequence of points defining a linear interpolated path
ST_MultiLineString data type	Stores a collection of ST_LineString data types
ST_MultiPoint data type	Stores a collection of ST_Point data types
ST_MultiPolygon data type	Stores a collection of ST_Polygon data types
ST_Point data type	Stores a zero-dimensional geometry that occupies a single location in coordinate space
ST_Polygon data type	Stores a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings
TimeSeries data type	Stores a collection of row subtypes

These extended data types of Informix are individually described in other topics. These data types are valid in local operations on databases where the data types are defined.

Extended data types in cross-database distributed SQL transactions

Distributed operations on other databases of the same Informix instance can access BOOLEAN, BLOB, CLOB, and LVARCHAR data types, which are implemented as built-in opaque types. Such operations can also access DISTINCT types whose base types are built-in types, and user-defined types (UDTs), if the UDTs and DISTINCT types are explicitly cast to built-in types, and if all of the UDTs, casts, and DISTINCT types are defined in all the participating databases.

You cannot, however, reference the following extended data types in cross-database transactions that access multiple databases of the local Informix instance:
- UDTs that are not cast to built-in data types
- DISTINCT types that are not cast to built-in data types
- Collection data types
- Named or unnamed ROW data types

Extended data types in cross-server distributed SQL transactions

Distributed SQL transactions and function calls that access databases of other Informix instances cannot return values of complex or smart large object data types, nor of most distinct or built-in opaque data types. Among the extended data types, only the following can be accessed in cross-server SQL operations:
- Any non-opaque built-in data type
- BOOLEAN
- DISTINCT of non-opaque built-in types
- DISTINCT of BOOLEAN
- DISTINCT of LVARCHAR
- DISTINCT of any of the DISTINCT types listed above
- IDSSECURITYLABEL
- LVARCHAR

A cross-server distributed SQL transaction can support DISTINCT data types only if they are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in each database that participates in the distributed operation. For queries or other DML operations in cross-server UDRs that use the data types in the preceding list as parameters or as returned data types, the UDR must also have the same definition in every participating database.

The built-in DISTINCT data type IDSSECURITYLABEL, which stores security label objects, can be accessed in cross-server and cross-database operations on protected data by users who hold sufficient security credentials. Like local operations on protected data, distributed queries that access remote tables protected by a security policy can return only the qualifying rows that IDSLBACRULES allow, after the database server has compared the security label that secures the data with the security credentials of the user who issues the query.

Related concepts:

"Extended Data Types" on page 2-46

Related reference:

"Built-In Data Types" on page 2-37

ANSI to Informix data type mapping

IBM Informix has equivalent data types to most ANSI data types.

The following table shows ANSI data types and the equivalent IBM Informix data types.

Table 2-3. Mapping of ANSI to Informix data types

ANSI data type	Informix data type
CHARACTER(n) or CHAR(n)	CHARACTER(n) or CHAR(n)
CHARACTER VARYING(n) or VARCHAR(n)	CHARACTER VARYING(n), VARCHAR(m,r), or LVARCHAR(n)
NATIONAL CHARACTER(n) or NCHAR(n)	NCHAR(n)
NATIONAL CHARACTER VARYING(n) or NVARCHAR(n)	NVARCHAR(m,r)
INTEGER	INTEGER or INT
SMALLINT	SMALLINT
FLOAT	FLOAT(n)
REAL	REAL or SMALLFLOAT
DOUBLE PRECISION	DOUBLE PRECISION or FLOAT(n)
NUMERIC(p,s) or DECIMAL(p,s)	NUMERIC(p,s) or DECIMAL(p,s)
DATE	DATE
TIMESTAMP	DATETIME YEAR TO FRACTION(n)

Description of Data Types

This section describes the data types that IBM Informix supports.

BIGINT data type

The BIGINT data type stores integers from -(2^{63} -1) to 2^{63} -1, which is –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807, in eight bytes.

This data type has storage advantages over INT8 and advantages for some arithmetic operations and sort comparisons over INT8 and DECIMAL data types.

BIGSERIAL data type

The BIGSERIAL data type stores a sequential integer, of the BIGINT data type, that is assigned automatically by the database server when a new row is inserted. The behavior of the BIGSERIAL data type is similar to the SERIAL data type, but with a larger range.

The default BIGSERIAL starting number is 1, but you can assign an initial value, *n*, when you create or alter the table. The value of *n* must be a positive integer in the range of 1 to 9,223,372,036,854,775,807. If you insert the value zero (0) in a BIGSERIAL column, the value that is used is the maximum positive value that already exists in the BIGSERIAL column + 1. If you insert any value that is not zero, that value will be inserted as it is.

A table can have no more than one SERIAL column, but it can have a SERIAL column and either a SERIAL8 column or a BIGSERIAL column.

For information about:

- The SERIAL data type, see “SERIAL(n) data type” on page 2-29

- Using the SERIAL8 data type with the INT8 or BIGINT data type, see “Using SERIAL8 and BIGSERIAL with INT8 or BIGINT”

Using SERIAL8 and BIGSERIAL with INT8 or BIGINT

All the arithmetic operators that are valid for INT8 and BIGINT (such as +, -, *, and /) and all the SQL functions that are valid for INT8 and BIGINT (such as ABS, MOD, POW, and so on) are also valid for SERIAL8 and BIGSERIAL values.

Data conversion rules that apply to INT8 and BIGINT also apply to SERIAL8 and BIGSERIAL, but with a NOT NULL constraint on SERIAL8 or BIGSERIAL.

The value of a SERIAL8 or BIGSERIAL column of one table can be stored in INT8 or BIGINT columns of another table. In the second table, however, the INT8 or BIGINT values are not subject to the constraints on the original SERIAL8 or BIGSERIAL column.

BLOB data type

The BLOB data type stores any kind of binary data in random-access chunks, called sbspaces. Binary data typically consists of saved spreadsheets, program-load modules, digitized voice patterns, and so on. The database server performs no interpretation of the contents of a BLOB column.

A BLOB column can be up to 4 terabytes ($4*2^{40}$ bytes) in length, though your system resources might impose a lower practical limit. The minimum amount of disk space allocated for smart large object data types is 512 bytes.

The term *smart large object* refers to BLOB and CLOB data types. Use CLOB data types (see page “CLOB data type” on page 2-11) for random access to text data. For general information about BLOB and CLOB data types, see “Smart large objects” on page 2-40.

You can use these SQL functions to perform operations on a BLOB column:
- **FILETOBLOB** copies a file into a BLOB column.
- **LOTOFILE** copies a BLOB (or CLOB) value into an operating-system file.
- **LOCOPY** copies an existing smart large object to a new smart large object.

For more information about these SQL functions, see the *IBM Informix Guide to SQL: Syntax*.

Within SQL, you are limited to the equality (=) comparison operation and the encryption and decryption functions for BLOB data. (The encryption and decryption functions are described in the *IBM Informix Guide to SQL: Syntax*.) To perform additional operations, you must use one of the application programming interfaces (APIs) from within your client application.

You can insert data into BLOB columns in the following ways:
- With the **dbload** or **onload** utilities
- With the LOAD statement (DB-Access)
- With the **FILETOBLOB** function
- From BLOB (**ifx_lo_t**) host variables (IBM Informix ESQL/C)

If you select a BLOB column using DB-Access, only the string `<SBlob value>` is returned; no actual value is displayed.

Related information:

FILETOBLOB and FILETOCLOB Functions

LOTOFILE Function

LOCOPY Function

BOOLEAN data type

The BOOLEAN data type stores TRUE or FALSE data values as a single byte.

The following table shows internal and literal representations of the BOOLEAN data type.

Logical Value	**Internal Representation**	**Literal Representation**
TRUE	\0	't'
FALSE	\1	'f'
NULL	Internal Use Only	NULL

You can compare two BOOLEAN values to test for equality or inequality. You can also compare a BOOLEAN value to the Boolean literals 't' and 'f'. BOOLEAN values are not case-sensitive; 't' is equivalent to 'T' and 'f' to 'F'.

You can use a BOOLEAN column to store what a Boolean expression returns. In the following example, the value of **boolean_column** is 't' if **column1** is less than **column2**, 'f' if **column1** is greater than or equal to **column2**, and NULL if the value of either **column1** or **column2** is unknown:

```
UPDATE my_table SET boolean_column = lessthan(column1, column2)
```

BYTE data type

The BYTE data type stores any kind of binary data in an undifferentiated byte stream. Binary data typically consists of digitized information, such as spreadsheets, program load modules, digitized voice patterns, and so on.

The term *simple large object* refers to an instance of a TEXT or BYTE data type. No more than 195 columns of the same table can be declared as BYTE and TEXT data types.

The BYTE data type has no maximum size. A BYTE column has a theoretical limit of 2^{31} bytes and a practical limit that your disk capacity determines.

You cannot use the MEDIUM or HIGH options of the UPDATE STATISTICS statement to calculate distribution statistics on BYTE columns.

BYTE objects in DML operations

You can store, retrieve, update, or delete the contents of a BYTE column. You cannot, however, use BYTE operands in arithmetic or string operations, nor assign literals to BYTE columns with the SET clause of the UPDATE statement. You also cannot use BYTE objects in any of the following contexts in a SELECT statement:
- With aggregate functions
- With the IN clause
- With the MATCHES or LIKE clauses
- With the GROUP BY clause
- With the ORDER BY clause

BYTE operands are valid in Boolean expressions only when you are testing for NULL values with the IS NULL or IS NOT NULL operators.

You can use the following methods, which can load rows or update fields, to insert BYTE data:
- With the **dbload** or **onload** utilities
- With the LOAD statement (DB-Access)
- From BYTE host variables (IBM Informix ESQL/C)

You cannot use a quoted text string, number, or any other actual value to insert or update BYTE columns.

When you select a BYTE column, you can receive all or part of it. To retrieve it all, use the regular syntax for selecting a column. You can also select any part of a BYTE column by using subscripts, as the next example, which reads the first 75 bytes of the **cat_picture** column associated with the catalog number 10001:

```
SELECT cat_picture [1,75] FROM catalog WHERE catalog_num = 10001
```

A built-in cast converts BYTE values to BLOB values. For more information, see the *IBM Informix Database Design and Implementation Guide*.

If you select a BYTE column using the DB-Access Interactive Schema Editor, only the string "`<BYTE value>`" is returned; no data value is displayed.

Important: If you try to return a BYTE column from a subquery, an error results, even if the column is not used in a Boolean expression nor with an aggregate.

CHAR(n) data type

The CHAR data type stores any string of letters, numbers, and symbols. It can store single-byte and multibyte characters, based on the database locale.

A CHAR(*n*) column has a length of *n* bytes, where 1 ≤ *n* ≤ 32,767. If you do not specify *n*, CHAR(1) is the default length. Character columns typically store alphanumeric strings, such as names, addresses, phone numbers, and so on. When a value is retrieved or stored as CHAR(*n*), exactly *n* bytes of data are transferred. If the string is shorter than *n* bytes, the string is extended with blank spaces up to the declared length. If the data value is longer than *n* bytes, a data string of length *n* that has been truncated from the right is inserted or retrieved, without the database server raising an exception.

This does not create partial characters in multibyte locales. In right-to-left locales, such as Arabic, Hebrew, or Farsi, the truncation is from the left.

Size specifications in CHAR data type declarations can be affected by the SQL_LOGICAL_CHAR feature that is described in the section “Logical Character Semantics in Character Type Declarations” on page 2-37.

For more information about East Asian locales that support multibyte code sets, see “Multibyte Characters with VARCHAR ” on page 2-36.

Treating CHAR Values as Numeric Values

If you plan to perform calculations on numbers stored in a column, you should assign a number data type to that column. Although you can store numbers in CHAR columns, you might not be able to use them in some arithmetic operations. For example, if you insert a sum into a CHAR column, you might experience overflow problems if the CHAR column is too small to hold the value. In this case, the insert fails. Numbers that have leading zeros (such as some zip codes) have the zeros stripped if they are stored as number types INTEGER or SMALLINT. Instead, store these numbers in CHAR columns.

Sorting and Relational Comparisons

In general, the collating order for sorting CHAR values is the order of characters in the code set. (An exception is the MATCHES operator with ranges; see "Collating VARCHAR Values" on page 2-36.) For more information about collation order, see the *IBM Informix GLS User's Guide*.

For multibyte locales, the database supports any multibyte characters in the code set. When storing multibyte characters in a CHAR data type, make sure to calculate the number of bytes needed. For more information about multibyte characters and locales, see the *IBM Informix GLS User's Guide*.

CHAR values are compared to other CHAR values by padding the shorter value on the right with blank spaces until the values have equal length, and then comparing the two values, using the code-set order for collation.

Nonprintable Characters with CHAR

A CHAR value can include tab, newline, whitespace, and nonprintable characters. You must, however, use an application to insert nonprintable characters into host variables and the host variables into your database. After passing nonprintable characters to the database server, you can store or retrieve them. After you select nonprintable characters, fetch them into host variables and display them with your own display mechanism.

An important exception is the first value in the ASCII code set is used as the end-of-data terminator symbol in columns of the CHAR data type. For this reason, any subsequent characters in the same string cannot be retrieved from a CHAR column, because the database server reads only the characters (if any) that precede this null terminator. For example, you cannot use the following 7-byte string as a CHAR data type value with a length of 7 bytes:

```
abc\0def
```

If you try to display nonprintable characters with DB-Access your screen returns inconsistent results. (Which characters are nonprintable is locale-dependent. For more information see the discussion of code-set conversion between the client and the database server in the *IBM Informix GLS User's Guide*.)

CHARACTER(n) data type

The CHARACTER data type is a synonym for CHAR.

CHARACTER VARYING(m,r) data type

The CHARACTER VARYING data type stores a string of letters, digits, and symbols of varying length, where *m* is the maximum size of the column (in bytes) and *r* is the minimum number of bytes reserved for that column.

The CHARACTER VARYING data type complies with ANSI/ISO standard for SQL; the non-ANSI VARCHAR data type supports the same functionality. For more information, see the description of the VARCHAR type in "VARCHAR(m,r) data type" on page 2-35.

CLOB data type

The CLOB data type stores any kind of text data in random-access chunks, called sbspaces. Text data can include text-formatting information, if this information is also textual, such as PostScript, Hypertext Markup Language (HTML), Standard Graphic Markup Language (SGML), or Extensible Markup Language (XML) data.

The term *smart large object* refers to CLOB and BLOB data types. The CLOB data type supports special operations for character strings that are inappropriate for BLOB values. A CLOB value can be up to 4 terabytes ($4*2^{40}$ bytes) in length. The minimum amount of disk space allocated for smart large object data types is 512 bytes.

Use the BLOB data type (see "BLOB data type" on page 2-7) for random access to binary data. For general information about the CLOB and BLOB data types, see "Smart large objects" on page 2-40.

The following SQL functions can perform operations on a CLOB column:

- **FILETOCLOB** copies a file into a CLOB column.
- **LOTOFILE** copies a CLOB (or BLOB) value into a file.
- **LOCOPY** copies a CLOB (or BLOB) value to a new smart large object.
- **ENCRYPT_DES** or **ENCRYPT_TDES** creates an encrypted BLOB value from a plain-text CLOB argument.
- **DECRYPT_BINAR** or **DECRYPT_CHAR** returns an unencrypted BLOB value from an encrypted BLOB argument (that **ENCRYPT_DES** or **ENCRYPT_TDES** created from a plain-text CLOB value).

For more information about these SQL functions, see the *IBM Informix Guide to SQL: Syntax*.

No casts exist for CLOB data. Therefore, the database server cannot convert data of the CLOB type to any other data type, except by using these encryption and decryption functions to return a BLOB. Within SQL, you are limited to the equality (=) comparison operation for CLOB data. To perform additional operations, you must use one of the application programming interfaces from within your client application.

Multibyte characters with CLOB

You can insert data into CLOB columns in the following ways:

- With the **dbload** or **onload** utilities
- With the LOAD statement (DB-Access)
- From CLOB (**ifx_lo_t**) host variables (ESQL/C)

For examples of CLOB types, see the *IBM Informix Guide to SQL: Tutorial* and the *IBM Informix Database Design and Implementation Guide*.

With GLS, the following rules apply:

- Multibyte CLOB characters must be defined in the database locale.
- The CLOB data type is collated in code-set order.
- The database server handles code-set conversions for CLOB data.

For more information about database locales, collation order, and code-set conversion, see the *IBM Informix GLS User's Guide*.

DATE data type

The DATE data type stores the calendar date. DATE data types require four bytes. A calendar date is stored internally as an integer value equal to the number of days since December 31, 1899.

Because DATE values are stored as integers, you can use them in arithmetic expressions. For example, you can subtract a DATE value from another DATE value. The result, a positive or negative INTEGER value, indicates the number of days that elapsed between the two dates. (You can use a UNITS DAY expression to convert the result to an INTERVAL DAY TO DAY data type.)

The following example shows the default display format of a DATE column:

```
mm/dd/yyyy
```

In this example, *mm* is the month (1-12), *dd* is the day of the month (1-31), and *yyyy* is the year (0001-9999). You can specify a different order of time units and a different time-unit separator than / (or no separator) by setting the **DBDATE** environment variable. For more information, see "DBDATE environment variable" on page 3-22.

In non-default locales, you can display dates in culture-specific formats. The locale and the **GL_DATE** and **DBDATE** environment variables (as described in the next chapter) affect the display formatting of DATE values. They do not, however, affect the internal storage format for DATE columns in the database. For more information, see the *IBM Informix GLS User's Guide*.

DATETIME data type

The DATETIME data type stores an instant in time expressed as a calendar date and time of day.

You select how precisely a DATETIME value is stored; its precision can range from a year to a fraction of a second.

DATETIME stores a data value as a contiguous series of fields that represents each time unit (*year*, *month*, *day*, and so forth) in the data type declaration.

Field qualifiers to specify a DATETIME data type have this format:

```
DATETIME largest_qualifier TO smallest_qualifier
```

This resembles an INTERVAL field qualifier, but DATETIME represents a point in time, rather than (like INTERVAL) a span of time. These differences exist between DATETIME and INTERVAL qualifiers:

- The DATETIME keyword replaces the INTERVAL keyword.
- DATETIME field qualifiers cannot specify a nondefault precision for the *largest_qualifier* time unit.
- Field qualifiers of a DATETIME data type can include YEAR, MONTH, and smaller time units, but an INTERVAL data type that includes the DAY field qualifier (or smaller time units) cannot also include the YEAR or MONTH field qualifiers.

The *largest_qualifier* and *smallest_qualifier* of a DATETIME data type can be any of the fields that the following table lists, provided that *smallest_qualifier* does not specify a larger time unit than *largest_qualifier*. (The largest and smallest time units can be the same; for example, DATETIME YEAR TO YEAR.)

Table 2-4. DATETIME field qualifiers

Qualifier field	Valid entries
YEAR	A year numbered from 1 to 9,999 (A.D.)
MONTH	A month numbered from 1 to 12
DAY	A day numbered from 1 to 31, as appropriate to the month
HOUR	An hour numbered from 0 (midnight) to 23
MINUTE	A minute numbered from 0 to 59
SECOND	A second numbered from 0 - 59
FRACTION	A decimal fraction-of-a-second with up to 5 digits of scale. The default scale is 3 digits (a thousandth of a second). For *smallest_qualifier to* specify another scale, write FRACTION(*n*), where *n* is the number of digits from 1 - 5.

The declaration of a DATETIME column need not include the full YEAR to FRACTION range of time units. It can include any contiguous subset of these time units, or even only a single time unit.

For example, you can enter a MONTH TO HOUR value in a column declared as YEAR TO MINUTE, if each entered value contains information for a contiguous series of time units. You cannot, however, enter a value for only the MONTH and HOUR; the entry must also include a value for DAY.

If you use the DB-Access TABLE menu, and you do not specify the DATETIME qualifiers, a default DATETIME qualifier, YEAR TO YEAR, is assigned.

A valid DATETIME literal must include the DATETIME keyword, the values to be entered, and the field qualifiers. You must include these qualifiers because, as noted earlier, the value that you enter can contain fewer fields than were declared for that column. Acceptable qualifiers for the first and last fields are identical to the list of valid DATETIME fields that are listed in the table Table 2-4.

Write values for the field qualifiers as integers and separate them with delimiters. The following table lists the delimiters that are used with DATETIME values in the default US English locale. (These are a superset of the delimiters that are used in INTERVAL values.)

Table 2-5. Delimiters used with DATETIME

Delimiter	Placement in DATETIME Literal
Hyphen (**-**)	Between the YEAR, MONTH, and DAY time-unit values
Blank space ( )	Between the DAY and HOUR time-unit values
Colon (**:**)	Between the HOUR, MINUTE, and SECOND time-unit values
Decimal point (**.**)	Between the SECOND and FRACTION time-unit values

The following illustration shows a DATETIME YEAR TO FRACTION(3) value with delimiters.

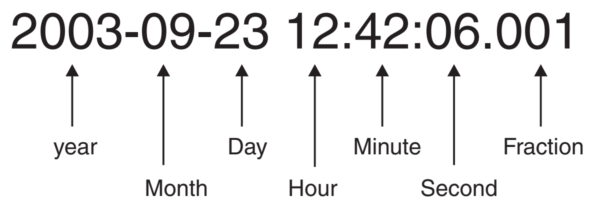


Figure 2-1. Example DATETIME Value with Delimiters

When you enter a value with fewer time-unit fields than in the column, the value that you enter is expanded automatically to fill all the declared time-unit fields. If you leave out any more significant fields, that is, time units larger than any that you include, those fields are filled automatically with the current values for those time units from the system clock calendar. If you leave out any less-significant fields, those fields are filled with zeros (or with 1 for MONTH and DAY) in your entry.

You can also enter DATETIME values as character strings. The character string must include information for each field defined in the DATETIME column. The INSERT statement in the following example shows a DATETIME value entered as a character string:

```
INSERT INTO cust_calls (customer_num, call_dtime, user_id,
      call_code, call_descr)
    VALUES (101, '2001-01-14 08:45', 'maryj', 'D',
      'Order late - placed 6/1/00');
```

If **call_dtime** is declared as DATETIME YEAR TO MINUTE, the character string must include values for the *year*, *month*, *day*, *hour*, and *minute* fields.

If the character string does not contain information for all the declared fields (or if it adds additional fields), then the database server returns an error.

All fields of a DATETIME column are two-digit numbers except for the *year* and *fraction* fields. The *year* field is stored as four digits. When you enter a two-digit value in the year field, how the abbreviated year is expanded to four digits depends on the setting of the **DBCENTURY** environment variable.

For example, if you enter `02` as the *year* value, whether the year is interpreted as `1902`, `2002`, or `2102` depends on the setting of **DBCENTURY** and on the value of the system clock calendar at execution time. If you do not set **DBCENTURY**, the leading digits of the current year are appended by default.

The *fraction* field requires *n* digits where 1 ≤ *n* ≤ 5, rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes that a DATETIME value requires:

```
(total number of digits for all fields) /2 + 1
```

For example, a YEAR TO DAY qualifier requires a total of eight digits (four for *year*, two for *month*, and two for *day*). According to the formula, this data value requires 5, or (8/2) + 1, bytes of storage.

The USEOSTIME configuration parameter can affect the subsecond granularity when the database server obtains the current time from the operating system in SQL statements. For details, see the *IBM Informix Administrator's Reference*.

With an ESQL API, the **DBTIME** environment variable affects DATETIME formatting. Nondefault locales and settings of the **GL_DATE** and **DBDATE** environment variables also affect the display of datetime data. They do not, however, affect the internal storage format of a DATETIME column.

If you specify a locale other than U.S. English, the locale defines the culture-specific display formats for DATETIME values. To change the default display format, change the setting of the **GL_DATETIME** environment variable. When a database with a nondefault locale uses a nondefault **GL_DATETIME** setting, the **USE_DTENV** environment variable must be set to 1 before the database server can correctly process localized DATETIME values in the following operations:

- using the LOAD or UNLOAD feature of DB-Access
- using the **dbexport** or **dbimport** migration utilities
- using DML statements of SQL on database tables or on objects that the CREATE EXTERNAL TABLE statement defined.

For more information about locales and GLS environment variables that can specify end-user DATETIME formats, see the *IBM Informix GLS User's Guide*.

Related concepts:

"Manipulating DATE with DATETIME and INTERVAL Values" on page 2-43

"Manipulating DATETIME Values" on page 2-42

Related reference:

"INTERVAL data type" on page 2-19

"DBCENTURY environment variable" on page 3-20

"DBTIME environment variable" on page 3-32

Related information:

The mi_datetime_compare() function

DEC data type

The DEC data type is a synonym for DECIMAL.

DECIMAL

The DECIMAL data type can take two forms: DECIMAL(*p*) floating point and DECIMAL(*p,s*) fixed point.

In an ANSI-compliant database all DECIMAL numbers are fixed point.

By default, literal numbers that include a decimal (**.**) point are interpreted by the database server as DECIMAL values.

DECIMAL(p) Floating Point

The DECIMAL data type stores decimal floating-point numbers up to a maximum of 32 significant digits, where *p* is the total number of significant digits (the *precision*).

Specifying precision is optional. If you specify no precision (*p*), DECIMAL is treated as DECIMAL(16), a floating-point decimal with a precision of 16 places. DECIMAL(*p*) has an absolute exponent range between 10^{-130} and 10^{124}.

If you declare a DECIMAL(*p*) column in an ANSI-compliant database, the scale defaults to DECIMAL(*p*, 0), meaning that only integer values can be stored in this data type.

In a database that is not ANSI-compliant, a DECIMAL(*p*) is a floating-point data type of a scale large enough to store the exponential notation for a value.

For example, the following calculation shows how many bytes of storage a DECIMAL(5) column requires in the default locale (where the decimal point occupies a single byte):

```
1 byte for the sign of the data value
1 byte for the 1st digit
1 byte for the decimal point
4 bytes for the rest of the digits (precision of 5 - 1)
1 byte for the e symbol
1 byte for the sign of the exponent
3 bytes for the exponent
------------------------------------------------
12 bytes total
```

Thus, "12345" in a DECIMAL(5) column is displayed as "12345.00000" (that is, with a scale of 6) in a database that is not ANSI-compliant.

DECIMAL (p,s) Fixed Point

In fixed-point numbers, DECIMAL(*p,s*), the decimal point is fixed at a specific place, regardless of the value of the number. When you specify a column of this type, you declare its precision (*p*) as the total number of digits that it can store, from 1 to 32. You declare its *scale* (*s*) as the total number of digits in the fractional part (that is, to the right of the decimal point).

All numbers with an absolute value less than $0.5 * 10^{-s}$ have the value zero. The largest absolute value of a DECIMAL(*p,s*) data type that you can store without an overflow error is $10^{p-s} -10^{-s}$. A DECIMAL column typically stores numbers with fractional parts that must be stored and displayed exactly (for example, rates or percentages). In an ANSI-compliant database, all DECIMAL numbers must have absolute values in the range 10^{-32} to 10^{+31}.

DECIMAL Storage

The database server uses one byte of disk storage to store two digits of a decimal number, plus an additional byte to store the exponent and sign, with the first byte representing a sign bit and a 7-bit exponent in excess-65 format. The rest of the bytes express the mantissa as base-100 digits. The significant digits to the left of the decimal and the significant digits to the right of the decimal are stored in separate groups of bytes. At the maximum *precision* specification, DECIMAL(32,s) data types can store s-1 decimal digits to the right of the decimal point, if s is an odd number.

How the database server stores decimal numbers is illustrated in the following example. If you specify DECIMAL(6,3), the data type consists of three significant digits in the integral part and three significant digits in the fractional part (for instance, 123.456). The three digits to the left of the decimal are stored on 2 bytes (where one of the bytes only holds a single digit) and the three digits to the right of the decimal are stored on another 2 bytes, as Figure 2-2 on page 2-17 illustrates.

(The exponent byte is not shown.) With the additional byte required for the exponent and sign, DECIMAL(6,3) requires a total of 5 bytes of storage.

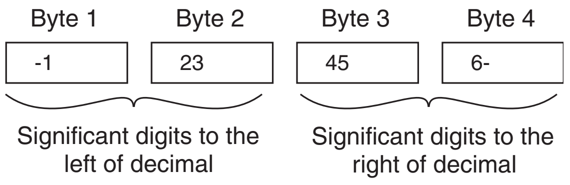

Byte 1
Byte 2
Byte 3
Byte 4
-1
23
45
6-
Significant digits to the left of decimal
Significant digits to the right of decimal

Figure 2-2. Schematic that illustrates the storage of digits in a decimal (p,s) value

You can use the following formulas (rounded down to a whole number of bytes) to calculate the byte storage (N) for a DECIMAL(*p,s*) data type (where N includes the byte that is required to store the exponent and the sign):

```
If the scale is odd: N = (precision + 4) / 2
If the scale is even: N = (precision + 3) / 2
```

For example, the data type DECIMAL(5,3) requires 4 bytes of storage (9/2 rounded down equals 4).

There is one caveat to these formulas. The maximum number of bytes the database server uses to store a decimal value is 17. One byte is used to store the exponent and sign, leaving 16 bytes to store up to 32 digits of precision. If you specify a precision of 32 and an *odd* scale, however, you lose 1 digit of precision. Consider, for example, the data type DECIMAL(32,31). This decimal is defined as 1 digit to the left of the decimal and 31 digits to the right. The 1 digit to the left of the decimal requires 1 byte of storage. This leaves only 15 bytes of storage for the digits to the right of the decimal. The 15 bytes can accommodate only 30 digits, so 1 digit of precision is lost.

DISTINCT data types

A DISTINCT type is a data type that is derived from a source type (called the base type).

A source type can be:
- A built-in type
- An existing DISTINCT type
- An existing named ROW type
- An existing opaque type

A DISTINCT type inherits from its source type the length and alignment on the disk. A DISTINCT type thus makes efficient use of the preexisting functionality of the database server.

When you create a DISTINCT data type, the database server automatically creates two explicit casts: one cast from the DISTINCT type to its source type and one cast from the source type to the DISTINCT type. A DISTINCT type based on a built-in source type does not inherit the built-in casts that are provided for the built-in type. A DISTINCT type does inherit, however, any user-defined casts that have been defined on the source type.

A DISTINCT type cannot be compared directly to its source type. To compare the two types, you must first explicitly cast one type to the other.

You must define a DISTINCT type in the database. Definitions of DISTINCT types are stored in the **sysxtdtypes** system catalog table. The following SQL statements maintain the definitions of DISTINCT types in the database:

- The CREATE DISTINCT TYPE statement adds a DISTINCT type to the database.
- The DROP TYPE statement removes a previously defined DISTINCT type from the database.

For more information about the SQL statements mentioned above, see the *IBM Informix Guide to SQL: Syntax*. For information about casting DISTINCT data types, see "Casts for distinct types" on page 2-53. For examples that show how to create and register cast functions for a DISTINCT type, see the *IBM Informix Database Design and Implementation Guide*.

Size specifications in declarations of DISTINCT types whose base types are built-in character types can be affected by the SQL_LOGICAL_CHAR feature that is described in the section "Logical Character Semantics in Character Type Declarations" on page 2-37.

DOUBLE PRECISION data types

The DOUBLE PRECISION keywords are a synonym for the FLOAT keyword.

Related reference:

"FLOAT(n)"

FLOAT(n)

The FLOAT data type stores double-precision floating-point numbers with up to 17 significant digits. FLOAT corresponds to IEEE 4-byte floating-point, and to the **double** data type in C. The range of values for the FLOAT data type is the same as the range of the C **double** data type on your computer.

You can use *n* to specify the precision of a FLOAT data type, but SQL ignores the precision. The value *n* must be a whole number between 1 and 14.

A column with the FLOAT data type typically stores scientific numbers that can be calculated only approximately. Because floating-point numbers retain only their most significant digits, the number that you enter in this type of column and the number the database server displays can differ slightly.

The difference between the two values depends on how your computer stores floating-point numbers internally. For example, you might enter a value of 1.1000001 into a FLOAT field and, after processing the SQL statement, the database server might display this value as 1.1. This situation occurs when a value has more digits than the floating-point number can store. In this case, the value is stored in its approximate form with the least significant digits treated as zeros.

FLOAT data types usually require 8 bytes of storage per value. Conversion of a FLOAT value to a DECIMAL value results in 17 digits of precision.

Related reference:

"DOUBLE PRECISION data types"

IDSSECURITYLABEL data type

The IDSSECURITYLABEL type stores a security label in a table that is protected by a label-based access control (LBAC) security policy.

Only a user who holds the **DBSECADM** role can create, alter, or drop a column of this data type. IDSSECURITYLABEL is a built-in DISTINCT OF VARCHAR(128) data type, but because its use is restricted to databases that implement label-based access control, it is not classified as a character data type. A table that is protected by a security policy can have only one IDSSECURITYLABEL column. A table that is not associated with any label-based security policy cannot include an IDSSECURITYLABEL column. You cannot encrypt the security label in a column of type IDSSECURITYLABEL.

For a discussion of security policies, security components, security labels, and other concepts of label-based access control (LBAC), see the IBM Informix Security Guide.

INT data type

The INT data type is a synonym for INTEGER.

INT8

The INT8 data type stores whole numbers that can range in value from –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 [or $-(2^{63}-1)$ to $2^{63}-1$], for 18 or 19 digits of precision.

The number –9,223,372,036,854,775,808 is a reserved value that cannot be used. The INT8 data type is typically used to store large counts, quantities, and so on.

IBM Informix stores INT8 data in internal format that can require up to 10 bytes of storage.

Arithmetic operations and sort comparisons are performed more efficiently on integer data than on floating-point or fixed-point decimal data, but INT8 cannot store data with absolute values beyond $| 2^{63}-1 |$. If a value exceeds the numeric range of INT8, the database server does not store the value.

INTEGER data type

The INTEGER data type stores whole numbers that range from -2,147,483,647 to 2,147,483,647 for 9 or 10 digits of precision.

The number 2,147,483,648 is a reserved value and cannot be used. The INTEGER value is stored as a signed binary integer and is typically used to store counts, quantities, and so on.

Arithmetic operations and sort comparisons are performed more efficiently on integer data than on float or decimal data. INTEGER columns, however, cannot store absolute values beyond $(2^{31}-1)$. If a data value lies outside the numeric range of INTEGER, the database server does not store the value.

INTEGER data types require 4 bytes of storage per value.

INTERVAL data type

The INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: *year-month intervals* and *day-time intervals*.

A year-month interval can represent a span of years and months, and a day-time interval can represent a span of days, hours, minutes, seconds, and fractions of a second.

An INTERVAL value is always composed of one value or a series of values that represents time units. Within a data-definition statement such as CREATE TABLE or ALTER TABLE that defines the precision of an INTERVAL data type, the qualifiers must have the following format:

```
INTERVAL largest_qualifier(n) TO smallest_qualifier
```

Here the *largest_qualifier* and *smallest_qualifier* keywords are taken from one of the two INTERVAL classes, as shown in the table Table 2-6.

If SECOND (or a larger time unit) is the *largest_qualifier,* the declaration of an INTERVAL data type can optionally specify *n*, the precision of the largest time unit (for *n* ranging from 1 to 9); this is not a feature of DATETIME data types.

If *smallest_qualifier* is FRACTION, you can also specify a scale in the range from 1 to 5. For FRACTION TO FRACTION qualifiers, the upper limit of *n* is 5, rather than 9. There are two incommensurable classes of INTERVAL data types:
- Those with a *smallest_qualifier* larger than DAY
- Those with a *largest_qualifier* smaller than MONTH

Table 2-6. Interval Classes

Interval Class	Time Units	Valid Entry
YEAR-MONTH INTERVAL	YEAR	A number of years
YEAR-MONTH INTERVAL	MONTH	A number of months
DAY-TIME INTERVAL	DAY	A number of days
DAY-TIME INTERVAL	HOUR	A number of hours
DAY-TIME INTERVAL	MINUTE	A number of minutes
DAY-TIME INTERVAL	SECOND	A number of seconds
DAY-TIME INTERVAL	FRACTION	A decimal fraction of a second, with up to 5 digits. The default scale is 3 digits (thousandth of a second). To specify a non-default scale, write FRACTION(*n*), where $1 \le n \le 5$.

As with DATETIME data types, you can define an INTERVAL to include only the subset of time units that you need. But because the construct of “month” (as used in calendar dates) is not a time unit that has a fixed number of days, a single INTERVAL value cannot combine months and days; arithmetic that involves operands of the two different INTERVAL classes is not supported.

A value entered into an INTERVAL column need not include the full range of time units that were specified in the data-type declaration of the column. For example, you can enter a value of HOUR TO SECOND precision into a column defined as DAY TO SECOND. A value must always consist, however, of contiguous time units. In the previous example, you cannot enter only the HOUR and SECOND values; you must also include MINUTE values.

A valid INTERVAL literal contains the INTERVAL keyword, the values to be entered, and the field qualifiers. (See the discussion of literal intervals in the *IBM Informix Guide to SQL: Syntax*.) When a value contains only one field, the largest and smallest fields are the same.

When you enter a value in an INTERVAL column, you must specify the largest and smallest fields in the value, just as you do for DATETIME values. In addition, you can optionally specify the precision of the first field (and the scale of the last field if it is a FRACTION). If the largest and smallest field qualifiers are both FRACTION, you can specify only the scale in the last field.

Acceptable qualifiers for the largest and smallest fields are identical to the list of INTERVAL fields that the tab;e Table 2-6 on page 2-20 displays.

If you use the DB-Access **TABLE** menu, but you specify no INTERVAL field qualifiers, then a default INTERVAL qualifier, YEAR TO YEAR, is assigned.

The *largest_qualifier* in an INTERVAL value can be up to nine digits (except for FRACTION, which cannot be more than five digits), but if the value that you want to enter is greater than the default number of digits allowed for that field, you must explicitly identify the number of significant digits in the value that you enter. For example, to define an INTERVAL of DAY TO HOUR that can store up to 999 days, you can specify it the following way:

```
INTERVAL DAY(3) TO HOUR
```

INTERVAL literals use the same delimiters as DATETIME literals (except that MONTH and DAY time units are not valid within the same INTERVAL value). the following table shows the INTERVAL delimiters.

Table 2-7. INTERVAL Delimiters

Delimiter	Placement in an INTERVAL Literal
Hyphen	Between the YEAR and MONTH portions of the value
Blank space	Between the DAY and HOUR portions of the value
Colon	Between the HOUR, MINUTE, and SECOND portions of the value
Decimal point	Between the SECOND and FRACTION portions of the value

You can also enter INTERVAL values as character strings. The character string must include information for the same time units that were specified in the data-type declaration for the column. The INSERT statement in the following example shows an INTERVAL value entered as a character string:

```
INSERT INTO manufact (manu_code, manu_name, lead_time)
   VALUES ('BRO', 'Ball-Racquet Originals', '160')
```

Because the **lead_time** column is defined as INTERVAL DAY(3) TO DAY, this INTERVAL value requires only one field, the span of days required for lead time. If the character string does not contain information for all fields (or adds additional fields), the database server returns an error. For additional information about entering INTERVAL values as character strings, see the *IBM Informix Guide to SQL: Syntax*.

By default, all fields of an INTERVAL column are two-digit numbers, except for the year and fraction fields. The year field is stored as four digits. The fraction field requires *n* digits where $1 \le n \le 5$, rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required for an INTERVAL value:

```
(total number of digits for all fields)/2 + 1
```

For example, INTERVAL YEAR TO MONTH requires six digits (four for *year* and two for *month*), and requires 4, or (6/2) + 1, bytes of storage.

For information about using INTERVAL as a constant expression, see the description of the INTERVAL Field Qualifier in the *IBM Informix Guide to SQL: Syntax*.

Related concepts:

"Manipulating DATE with DATETIME and INTERVAL Values" on page 2-43

"Manipulating INTERVAL Values" on page 2-45

Related reference:

"DATETIME data type" on page 2-12

Related information:

The mi_interval_compare() function

LIST(e) data type

The LIST data type is a collection type that can store ordered non-NULL elements of the same SQL data type.

The LIST data type supports, but does not require, duplicate element values. The elements of a LIST data type have ordinal positions. The LIST object must have a first element, which can be followed by a second element, and so on.

For unordered collection data types that do not support ordinal positions, see "MULTISET(e) data type" on page 2-25 and "SET(e) data type" on page 2-31. For complex data types that can store a set of values that includes different SQL data types, see "ROW Data Types" on page 2-48.

No more than 97 columns of the same table can be declared as LIST data types. (The same restriction applies to SET and MULTISET collection types.)

By default, the database server inserts new elements into a LIST object at the end of the set of elements. To support the ordinal position of a LIST, the INSERT statement provides the AT clause. This clause allows you to specify the position at which you want to insert a LIST element value. For more information, see the INSERT statement in the *IBM Informix Guide to SQL: Syntax*.

All elements in a LIST object have the same element type. To specify the element type, use the following syntax:

```
LIST(element_type NOT NULL)
```

The *element_type* of a LIST can be any of the following data types:

- A built-in type, except SERIAL, SERIAL8, BIGSERIAL, BYTE, and TEXT
- A DISTINCT type
- An unnamed or named ROW type
- Another collection type
- An opaque type

You must specify the NOT NULL constraint for LIST elements. No other constraints are valid for LIST columns. For more information about the syntax of the LIST data type, see the *IBM Informix Guide to SQL: Syntax*.

You can use LIST in most contexts where any other data type is valid. For example:

- After the IN predicate in the WHERE clause of a SELECT statement to search for matching LIST values
- As an argument to the CARDINALITY or **mi_collection_card( )** function to determine the number of elements in a LIST column

You *cannot* use LIST values as arguments to an aggregate function such as AVG, MAX, MIN, or SUM.

Just as with the other collection data types, you must use parentheses (( )) in data type declarations to delimit the set of elements of a LIST data type:

```
CREATE FUNCTION  update_nums( list1 LIST (ROW (a VARCHAR(10),
                                               b VARCHAR(10),
                                               c INT) NOT NULL ));
```

In SQL expressions that include literal LIST values, however, you must use braces ({ }) to delimit the set of elements of a LIST object, as in the examples that follow.

Two LIST values are equal if they have the same elements in the same order. The following are both examples of LIST objects, but their values are not equal. :

```
LIST{"blue", "green", "yellow"}
```

```
LIST{"yellow", "blue", "green"}
```

The above expressions are not equal because the values are not in the same order. To be equal, the second statement must be:

```
LIST{"blue", "green", "yellow"}
```

LVARCHAR(m) data type

Use the LVARCHAR data type to create a column for storing variable-length character strings whose upper limit (*m*) can be up to 32,739 bytes.

This limit is much greater than the VARCHAR data type, which is used for character strings that are no longer than 255 bytes.

The LVARCHAR data type is implemented as a built-in opaque data type. You can access LVARCHAR columns in remote tables by using distributed queries across databases of the same or different IBM Informix instances.

By default, the database server interprets quoted strings as LVARCHAR types. It also uses LVARCHAR 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 database servers can manipulate. Each opaque type also has an output support function and cast, which convert it from its internal representation to LVARCHAR.

Important: When LVARCHAR is declared (with no size specification) as the data type of a column in a database table, the default maximum size is 2 KB (2048 bytes), but you can specify an explicit maximum length of up to 32,739 bytes. When LVARCHAR is used in I/O operations on an opaque data type, however, the maximum size is limited only by the operating system.

You cannot use the MEDIUM or HIGH options of the UPDATE STATISTICS statement to calculate distribution statistics on LVARCHAR columns.

Size specifications in LVARCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR feature that is described in the section "Logical Character Semantics in Character Type Declarations" on page 2-37.

For more information about LVARCHAR, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

MONEY(p,s) data type

The MONEY data type stores currency amounts.

TLike the DECIMAL(*p*,*s*) data type, MONEY can store fixed-point numbers up to a maximum of 32 significant digits, where *p* is the total number of significant digits (the precision) and *s* is the number of digits to the right of the decimal point (the scale).

Unlike the DECIMAL data type, the MONEY data type is always treated as a fixed-point decimal number. The database server defines the data type MONEY(*p*) as DECIMAL(*p*,2). If the precision and scale are not specified, the database server defines a MONEY column as DECIMAL(16,2).

You can use the following formula (rounded down to a whole number of bytes) to calculate the byte storage for a MONEY data type:

```
If the scale is odd: N = (precision + 4) / 2
If the scale is even: N = (precision + 3) / 2
```

For example, a MONEY data type with a precision of 16 and a scale of 2 (MONEY(16,2)) requires 10 or (16 + 3)/2, bytes of storage.

In the default locale, client applications format values from MONEY columns with the following currency notation:

- A currency symbol: a dollar sign (**$**) at the front of the value
- A thousands separator: a comma (**,**) that separates every three digits in the integer part of the value
- A decimal point: a period (**.**) between the integer and fractional parts of the value

To change the format for MONEY values, change the **DBMONEY** environment variable. For valid **DBMONEY** settings, see "DBMONEY environment variable" on page 3-27.

The default value that the database server uses for scale is locale-dependent. The default locale specifies a default scale of two. For non-default locales, if the scale is omitted from the declaration, the database server creates MONEY values with a locale-specific scale.

The currency notation that client applications use is locale-dependent. If you specify a nondefault locale, the client uses a culture-specific format for MONEY values that might differ from the default U.S. English format in the leading (or trailing) currency symbol, thousands separator, and decimal separator, depending on what the locale files specify. For more information about locale dependency, see the *IBM Informix GLS User's Guide*.

MULTISET(e) data type

The MULTISET data type is a collection type that stores a non-ordered set that can include duplicate element values.

The elements in a MULTISET have no ordinal position. That is, there is no concept of a first, second, or third element in a MULTISET. (For a collection type with ordinal positions for elements, see "LIST(e) data type" on page 2-22.)

All elements in a MULTISET have the same element type. To specify the element type, use the following syntax:

```
MULTISET(element_type NOT NULL)
```

The *element_type* of a collection can be any of the following types:

- Any built-in type, except SERIAL, SERIAL8, BIGSERIAL, BYTE, and TEXT
- An unnamed or a named ROW type
- Another collection type or opaque type

You can use MULTISET anywhere that you use any other data type, unless otherwise indicated. For example:

- After the IN predicate in the WHERE clause of a SELECT statement to search for matching MULTISET values
- As an argument to the CARDINALITY or **mi_collection_card( )** function to determine the number of elements in a MULTISET column

You *cannot* use MULTISET values as arguments to an aggregate function such as AVG, MAX, MIN, or SUM.

You must specify the NOT NULL constraint for MULTISET elements. No other constraints are valid for MULTISET columns. For more information about the MULTISET collection type, see the *IBM Informix Guide to SQL: Syntax*.

Two multiset data values are equal if they have the same elements, even if the elements are in different positions within the set. The following examples are both multiset values but are not equal:

```
MULTISET {"blue", "green", "yellow"}
MULTISET {"blue", "green", "yellow", "blue"}
```

The following multiset values are equal:

```
MULTISET {"blue", "green", "blue", "yellow"}
MULTISET {"blue", "green", "yellow", "blue"}
```

No more than 97 columns of the same table can be declared as MULTISET data types. (The same restriction applies to SET and LIST collection types.)

Named ROW

See "ROW data type, Named" on page 2-27.

NCHAR(n) data type

The NCHAR data type stores fixed-length character data. The data can be a string of single-byte or multibyte letters, digits, and other symbols that are supported by the code set of the database locale.

The main difference between CHAR and NCHAR data types is the collating order.

The collation order of the CHAR data type follows the code-set order, but the collating order of the NCHAR data type can be a localized order, if **DB_LOCALE** (or SET COLLATION) specifies a locale that defines a localized order for collation.

Size specifications ib NCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR configuration parameter that is described in the section "Logical Character Semantics in Character Type Declarations" on page 2-37.

In databases that are created with the NLSCASE INSENSITIVE property, operations on NCHAR strings ignore letter case, ordering data values without respect to or preference for letter case. For example, the NCHAR string `"IDS"` might precede or follow `"IdS"` or `"iDs"` in the collated list that a query returns, depending on the order in which these data strings are retrieved, because all of the following NCHAR strings are treated as duplicate values:

```
"ids" "IDS" "idS" "IDs" "IdS" "iDs" "iDS" "Ids"
```

NUMERIC(p,s) data type

The NUMERIC data type is a synonym for fixed-point DECIMAL.

NVARCHAR(m,r) data type

The NVARCHAR data type stores strings of varying lengths. The string can include digits, symbols, and both single-byte and (in some locales) multibyte characters.

The main difference between VARCHAR and NVARCHAR data types is the collation order. Collation of VARCHAR data follows code-set order, but NVARCHAR collation can be locale specific, if **DB_LOCALE** (or SET COLLATION) has specified a locale that defines a localized order for collation. (The section "Collating VARCHAR Values" on page 2-36 describes an exception.)

A column declared as NVARCHAR, without parentheses or parameters, has a maximum size of one byte, and a reserved size of zero.

The first parameter in NVARCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR configuration parameter that is described in the section "Logical Character Semantics in Character Type Declarations" on page 2-37.

No more than 195 columns of the same table can be NVARCHAR data types.

In databases that are created with the NLSCASE INSENSITIVE property, operations on NVARCHAR strings ignore letter case, ordering data values without respect to or preference for letter case. For example, the NVARCHAR string `"IBM"` might precede or follow `"IbM"` or `"iBm"` in the collated list that a query returns, depending on the order in which these data strings are retrieved, because all of the following NVARCHAR strings are treated as duplicate values:

```
"ibm" "IBM" "ibM" "IBm" "IbM" "iBm" "iBM" "Ibm"
```

OPAQUE data types

An OPAQUE type is a data type for which you must provide information to the database server.

You must provide this information:

- A data structure for how the data values are stored on disk

- Support functions to determine how to convert between the disk storage format and the user format for data entry and display
- Secondary access methods that determine how the index on this data type is built, used, and manipulated
- User functions that use the data type
- A system catalog entry to register the OPAQUE type in the database

The internal structure of an OPAQUE type is not visible to the database server and can only be accessed through user-defined routines. Definitions for OPAQUE types are stored in the **sysxtdtypes** system catalog table. These SQL statements maintain the definitions of OPAQUE types in the database:

- The CREATE OPAQUE TYPE statement registers a new OPAQUE type in the database.
- The DROP TYPE statement removes a previously defined OPAQUE type from the database.

For more information about the above-mentioned SQL statements, see the *IBM Informix Guide to SQL: Syntax*. For information about how to create OPAQUE types and an example of an OPAQUE type, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

REAL data type

The REAL data type is a synonym for SMALLFLOAT.

ROW data type, Named

A named ROW data type must be declared with a name. This SQL identifier must be unique among data type names within the same database.

(An unnamed ROW type is a ROW type that contains fields but has no user-defined name.) Only named ROW types support data type inheritance. For more information, see "ROW Data Types" on page 2-48.

Defining named ROW types

You must declare and register in the database a new named ROW type by using the CREATE ROW TYPE statement of SQL. Definitions for named ROW types are stored in the **sysxtdtypes** system catalog table.

The fields of a ROW data type can be any built-in data type or UDT, but TEXT or BYTE fields of a ROW type are valid in typed tables only. If you want to assign a ROW type to a column in the CREATE TABLE or ALTER TABLE statements, its elements cannot be TEXT or BYTE data types.

In general, the data type of a field of a ROW type can be any of these types:

- A built-in type (except for the TEXT or BYTE data types)
- A collection type (LIST, MULTISET, or SET)
- A distinct type
- Another named or unnamed ROW type
- An opaque type

These SQL statements maintain the definitions of named ROW data types:

- The CREATE ROW TYPE statement adds a named ROW type to the database.

- The DROP ROW TYPE statement removes a previously defined named ROW type from the database.

No more than 195 columns of the same table can be named ROW types.

For details about these SQL syntax statements, see the *IBM Informix Guide to SQL: Syntax*. For examples of how to create and use named ROW types, see the *IBM Informix Database Design and Implementation Guide*.

Equivalence and named ROW types

No two named ROW types can be equal, even if they have identical structures, because they have different names. For example, the following named ROW types have the same structure (the same number of fields and the same order of data types of fields within the row) but they are not equal:

```
name_t (lname CHAR(15), initial CHAR(1), fname CHAR(15))
emp_t (lname CHAR(15), initial CHAR(1), fname CHAR(15))
```

A Boolean equality condition like `name_t = emp_t` always evaluates to FALSE if both of the operands are different named ROW types.

Named ROW types and inheritance

Named ROW types can be part of a type-inheritance hierarchy. One named ROW type can be the parent (or supertype) of another named ROW type. A subtype in a hierarchy inherits all the properties of its supertype. Type inheritance is explained in the CREATE ROW TYPE statement in the *IBM Informix Guide to SQL: Syntax* and in the *IBM Informix Database Design and Implementation Guide*.

Typed tables

Tables that are part of an inheritance hierarchy must be typed tables. Typed tables are tables that have been assigned a named ROW type. For the syntax you use to create typed tables, see the CREATE TABLE statement in the *IBM Informix Guide to SQL: Syntax*. Table inheritance and its relation to type inheritance is also explained in that section. For information about how to create and use typed tables, see the *IBM Informix Database Design and Implementation Guide*.

ROW data type, Unnamed

An unnamed ROW type contains fields but has no user-declared name. An unnamed ROW type is defined by its structure.

Two unnamed ROW types are equal if they have the same structure (meaning the ordered list of the data types of the fields). If two unnamed ROW types have the same number of fields, and if the order of the data type of each field in one ROW type matches the order of data types of the corresponding fields in the other ROW data type, then the two unnamed ROW data types are equal.

For example, the following unnamed ROW types are equal:

```
ROW (lname char(15), initial char(1) fname char(15))
ROW (dept char(15), rating char(1) name char(15))
```

The following ROW types have the same number of fields and the same data types, but are not equal, because their fields are not in the same order:

```
ROW (x integer, y varchar(20), z real)
ROW (x integer, z real, y varchar(20))
```

A field of an unnamed ROW type can be any of the following data types:
- A built-in type
- A collection type
- A distinct type
- Another ROW type
- An opaque type

Unnamed ROW types cannot be used in typed tables or in type inheritance hierarchies. For more information about unnamed ROW types, see the *IBM Informix Guide to SQL: Syntax* and the *IBM Informix Database Design and Implementation Guide*.

Creating unnamed ROW types

You can create an unnamed ROW type in several ways:
- You can declare an unnamed ROW type using the ROW keyword. Each field in a ROW can have a different field type. To specify the field type, use the following syntax:

  `ROW(field_name field_type, ...)`

  The *field_name* must conform to the rules for SQL identifiers. (See the Identifier section in the *IBM Informix Guide to SQL: Syntax*.)
- To generate an unnamed ROW type, use the ROW keyword as a constructor with a series of values. A corresponding unnamed ROW type is created, using the default data types of the specified values.

  For example, the following declaration:

  ```
  ROW(1, 'abc', 5.30)
  ```

  defines this unnamed ROW data type:

  ```
  ROW (x INTEGER, y VARCHAR, z DECIMAL)
  ```
- You can create an unnamed ROW type by an implicit or explicit cast from a named ROW type or from another unnamed ROW type.
- The rows of any table (except a table defined on a named ROW type) are unnamed ROW types.

No more than 195 columns of the same table can be unnamed ROW types.

Inserting Values into Unnamed ROW Type Columns

When you specify field values for an unnamed ROW type, list the field values after the constructor and between parentheses. For example, suppose you have an unnamed ROW-type column. The following INSERT statement adds one group of field values to this ROW column:

```
INSERT INTO table1 VALUES (ROW(4, 'abc'))
```

You can specify a ROW column in the IN predicate in the WHERE clause of a SELECT statement to search for matching ROW values. For more information, see the Condition section in the *IBM Informix Guide to SQL: Syntax*.

SERIAL(n) data type

The SERIAL data type stores a sequential integer, of the INT data type, that is automatically assigned by the database server when a new row is inserted.

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table.

- You must specify a positive number for the starting number.
- If you specify zero (0) for the starting number, the value that is used is the maximum positive value that already exists in the SERIAL column + 1.

The maximum value for SERIAL is 2,147,483,647. If you assign a number greater than 2,147,483,647, you receive a syntax error. Use the SERIAL8 or BIGSERIAL data type, rather than SERIAL, if you need a larger range.

A table can have no more than one SERIAL column, but it can have a SERIAL column and either a SERIAL8 column or a BIGSERIAL column.

SERIAL values in a column are not automatically unique. You must apply a unique index or primary key constraint to this column to prevent duplicate serial numbers. If you use the interactive schema editor in DB-Access to define the table, a unique index is applied automatically to a SERIAL column.

SERIAL numbers might not be consecutive, because of concurrent users, rollbacks, and other factors.

The DEFINE *variable* LIKE *column* syntax of SPL for indirect typing declares a variable of the INTEGER data type if *column* is a SERIAL data type.

After a number is assigned, it cannot be changed. You can insert a value into a SERIAL column (using the INSERT statement) or reset a serial column (using the ALTER TABLE statement), if the new value does not duplicate any existing value in the column. To insert into a SERIAL column, your database server increments by one the previous value (or the reset value, if that is larger) and assigns the result as the entered value. If ALTER TABLE has reset the next value of a SERIAL column to a value smaller than values already in that column, however, the next value follows this formula:

```
(maximum existing value in SERIAL column) + 1
```

For example, if you reset the serial value of **customer.customer_num** to `50`, when the largest existing value is `128`, the next assigned number will be `129`. For more details on SERIAL data entry, see the *IBM Informix Guide to SQL: Syntax*.

A SERIAL column can store unique codes such as order, invoice, or customer numbers. SERIAL data values require four bytes of storage, and have the same precision as the INTEGER data type. For details of another way to assign unique whole numbers to each row of a database table, see the CREATE SEQUENCE statement in *IBM Informix Guide to SQL: Syntax*.

SERIAL8(n) data type

The SERIAL8 data type stores a sequential integer, of the INT8 data type, that is assigned automatically by the database server when a new row is inserted.

The SERIAL8 data type behaves like the SERIAL data type, but with a larger range. For more information about how to insert values into SERIAL8 columns, see the *IBM Informix Guide to SQL: Syntax*.

A SERIAL8 data column is commonly used to store large, unique numeric codes such as order, invoice, or customer numbers. SERIAL8 data values have the same precision and storage requirements as INT8 values (page "INT8" on page 2-19).

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table.

- You must specify a positive number for the starting number.
- If you specify zero (0) for the starting number, the value that is used is the maximum positive value that already exists in the SERIAL8 column + 1.

A table can have no more than one SERIAL column, but it can have a SERIAL column and either a SERIAL8 column or a BIGSERIAL column.

SERIAL8 values in a column are not automatically unique. You must apply a unique index or primary key constraint to this column to prevent duplicate serial numbers. If you use the interactive schema editor in DB-Access to define the table, a unique index is applied automatically to a SERIAL8 column.

SERIAL8 numbers might not be consecutive, because of concurrent users, rollbacks, and other factors.

The DEFINE *variable* LIKE *column* syntax of SPL for indirect typing declares a variable of the INTEGER data type if *column* is a SERIAL8 data type.

For more information, see “Assigning a Starting Value for SERIAL8.” For information about using the SERIAL8 data type with the INT8 or BIGINT data type, see “Using SERIAL8 and BIGSERIAL with INT8 or BIGINT” on page 2-7

Assigning a Starting Value for SERIAL8

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table. To start the values at `1` in a SERIAL8 column of a table, give the value `0` for the SERIAL8 column when you insert rows into that table. The database server will assign the value `1` to the SERIAL8 column of the first row of the table. The largest SERIAL8 value that you can assign is 2^{63}-1 (9,223,372,036,854,775,807). If you assign a value greater than this, you receive a syntax error. When the database server generates a SERIAL8 value of this maximum number, it wraps around and starts generating values beginning at 1.

After a nonzero SERIAL8 number is assigned, it cannot be changed. You can, however, insert a value into a SERIAL8 column (using the INSERT statement) or reset the SERIAL8 value *n* (using the ALTER TABLE statement), if that value does not duplicate any existing values in the column.

When you insert a number into a SERIAL8 column or reset the next value of a SERIAL8 column, your database server assigns the next number in sequence to the number entered. If you reset the next value of a SERIAL8 column to a value that is less than the values already in that column, however, the next value is computed using the following formula:

```
maximum existing value in SERIAL8 column + 1
```

For example, if you reset the SERIAL8 value of the **customer_num** column in the **customer** table to `50`, when the highest-assigned customer number is 128, the next customer number assigned is 129.

For information about using the SERIAL8 data type with the INT8 or BIGINT data type, see “Using SERIAL8 and BIGSERIAL with INT8 or BIGINT” on page 2-7

SET(e) data type

The SET data type is an unordered collection type that stores unique elements

Duplicate element values are not valid as explained in *IBM Informix Guide to SQL: Syntax*. (For a collection type that supports duplicate values, see the description of MULTISET in "MULTISET(e) data type" on page 2-25.)

No more than 97 columns of the same table can be declared as SET data types. (The same restriction also applies to MULTISET and LIST collection types.)

The elements in a SET have no ordinal position. That is, no construct of a first, second, or third element in a SET exists. (For a collection type with ordinal positions for elements, see "LIST(e) data type" on page 2-22.) All elements in a SET have the same element type. To specify the element type, use this syntax:

```
SET(element_type NOT NULL)
```

The *element_type* of a collection can be any of the following types:

- A built-in type, except SERIAL, SERIAL8, BIGSERIAL, BYTE, and TEXT
- A named or unnamed ROW type
- Another collection type
- An opaque type

You must specify the NOT NULL constraint for SET elements. No other constraints are valid for SET columns. For more information about the syntax of the SET collection type, see the *IBM Informix Guide to SQL: Syntax*.

You can use SET anywhere that you use any other data type, unless otherwise indicated. For example:

- After the IN predicate in the WHERE clause of a SELECT statement to search for matching SET values
- As an argument to the CARDINALITY or **mi_collection_card( )** function to determine the number of elements in a SET column

SET values are not valid as arguments to an aggregate function such as AVG, MAX, MIN, or SUM. For more information, see the Condition and Expression sections in the *IBM Informix Guide to SQL: Syntax*.

The following examples declare two sets. The first statement declares a set of integers and the second declares a set of character elements.

```
SET(INTEGER NOT NULL)
SET(CHAR(20) NOT NULL)
```

The following examples construct the same sets from value lists:

```
SET{1, 5, 13}
SET{"Oakland", "Menlo Park", "Portland", "Lenexa"}
```

In the following example, a SET constructor function is part of a CREATE TABLE statement:

```
CREATE TABLE tab
(
   c CHAR(5),
   s SET(INTEGER NOT NULL)
);
```

The following set values are equal:

```
SET{"blue", "green", "yellow"}
SET{"yellow", "blue", "green"}
```

SMALLFLOAT

The SMALLFLOAT data type stores single-precision floating-point numbers with approximately nine significant digits.

SMALLFLOAT corresponds to the **float** data type in C. The range of values for a SMALLFLOAT data type is the same as the range of values for the C **float** data type on your computer.

A SMALLFLOAT data type column typically stores scientific numbers that can be calculated only approximately. Because floating-point numbers retain only their most significant digits, the number that you enter in this type of column and the number the database displays might differ slightly depending on how your computer stores floating-point numbers internally.

For example, you might enter a value of 1.1000001 in a SMALLFLOAT field and, after processing the SQL statement, the application might display this value as 1.1. This difference occurs when a value has more digits than the floating-point number can store. In this case, the value is stored in its approximate form with the least significant digits treated as zeros.

SMALLFLOAT data types usually require four bytes of storage. Conversion of a SMALLFLOAT value to a DECIMAL value results in nine digits of precision.

SMALLINT data type

The SMALLINT data type stores small whole numbers that range from –32,767 to 32,767. The maximum negative number, –32,768, is a reserved value and cannot be used.

The SMALLINT value is stored as a signed binary integer.

Integer columns typically store counts, quantities, and so on. Because the SMALLINT data type requires only two bytes per value, arithmetic operations are performed efficiently. SMALLINT, however, stores only a limited range of values, compared to other built-in numeric data types. If a number is outside the range of the minimum and maximum SMALLINT values, the database server does not store the data value, but instead issues an error message.

TEXT data type

The TEXT data type stores any kind of text data. It can contain both single-byte and multibyte characters that the locale supports. The term *simple large object* refers to an instance of a TEXT or BYTE data type.

A TEXT column has a theoretical limit of 2^{31} bytes (two gigabytes) and a practical limit that your available disk storage determines. No more than 195 columns of the same table can be declared as TEXT data types. The same restriction also applies to BYTE data types.

You can store, retrieve, update, or delete the value in a TEXT column.

You can use TEXT operands in Boolean expressions only when you are testing for NULL values with the IS NULL or IS NOT NULL operators.

You can use the following methods, which can load rows or update fields, to insert TEXT data:

- With the **dbload** or **onload** utilities
- With the LOAD statement (DB-Access)
- From TEXT host variables (ESQL)

A built-in cast exists to convert TEXT objects to CLOB objects. For more information, see the *IBM Informix Database Design and Implementation Guide*.

Strings of the TEXT data type are collated in code-set order. For more information about collating orders, see the *IBM Informix GLS User's Guide*.

Selecting data in a TEXT column

When you select a TEXT column, you can receive all or part of it. To retrieve it all, use the regular syntax for selecting a column. You can also select any part of a TEXT column by using subscripts, as this example shows:

```
SELECT cat_descr [1,75] FROM catalog WHERE catalog_num = 10001
```

The SELECT statement reads the first 75 bytes of the **cat_descr** column associated with the **catalog_num** value `10001`.

Loading data into a TEXT column

You can use the LOAD statement to insert data into a table. For example, the `inp.txt` file contains the following information:

```
1|aaaaa|
2|bbbbb|
3|cccccc|
```

To load this data into the `blobtab` table use the following statement:

```
LOAD FROM inp.txt INSERT INTO blobtab;
```

Limitations

You cannot use TEXT operands in arithmetic or string expressions, nor can you assign literals to TEXT columns in the SET clause of the UPDATE statement.

You also cannot use TEXT values in any of the following ways:

- With aggregate functions
- With the IN clause
- With the MATCHES or LIKE clauses
- With the GROUP BY clause
- With the ORDER BY clause

You cannot use a quoted text string, number, or any other actual value to insert or update TEXT columns.

You cannot use the MEDIUM or HIGH options of the UPDATE STATISTICS statement to calculate distribution statistics on TEXT columns.

Important: An error results if you try to return a TEXT column from a subquery, even if no TEXT column is used in a comparison condition or with the IN predicate.

Nonprintable Characters in TEXT Values

TEXT columns typically store documents, program source files, and so on. In the default U.S. English locale, data objects of type TEXT can contain a combination of printable ASCII characters and the following control characters:

- Tab (CTRL-I)
- New line (CTRL-J)
- New page (CTRL-L)

Both printable and nonprintable characters can be inserted in text columns. IBM Informix products do not do any checking of data values that are inserted in a column of the TEXT data type. (Applications might have difficulty, however, in displaying TEXT values that include non-printable characters.) For detailed information about entering and displaying nonprintable characters, see "Nonprintable Characters with CHAR" on page 2-10.

Unnamed ROW

See "ROW data type, Unnamed" on page 2-28.

VARCHAR(m,r) data type

The VARCHAR data type stores character strings of varying length that contain single-byte and (if the locale supports them) multibyte characters, where *m* is the maximum size (in bytes) of the column and *r* is the minimum number of bytes reserved for that column.

A column declared as VARCHAR without parentheses or parameters has a maximum size of one byte, and a reserved size of zero.

The VARCHAR data type is the IBM Informix implementation of a character varying data type. The ANSI standard data type for varying-length character strings is CHARACTER VARYING.

The size of the maximum size (*m*) parameter of a VARCHAR column can range from 1 to 255 bytes. If you are placing an index on a VARCHAR column, the maximum size is 254 bytes. You can store character strings that are shorter, but not longer, than the *m* value that you specify.

Specifying the minimum reserved space (*r*) parameter is optional. This value can range from `0` to `255` bytes but must be less than the maximum size (*m*) of the VARCHAR column. If you do not specify any minimum value, it defaults to 0. You should specify this parameter when you initially intend to insert rows with short or NULL character strings in the column but later expect the data to be updated with longer values.

For variable-length strings longer than 255 bytes, you can use the LVARCHAR data type, whose upper limit is 32,739 bytes, instead of VARCHAR.

In an index based on a VARCHAR column (or on a NVARCHAR column), each index key has a length that is based on the data values that are actually entered, rather than on the declared maximum size of the column. (See, however, "IFX_PAD_VARCHAR environment variable" on page 3-46 for information about how you can configure the effective size of VARCHAR and NVARCHAR data strings that IBM Informix sends or receives.)

When you store a string in a VARCHAR column, only the actual data characters are stored. The database server does not strip a VARCHAR string of any user-entered trailing blanks, nor pad a VARCHAR value to the declared length of the column. If you specify a reserved space (*r*), but some data strings are shorter than *r* bytes, some space reserved for rows goes unused.

VARCHAR values are compared to other VARCHAR values (and to other character-string data types) in the same way that CHAR values are compared. The shorter value is padded on the right with blank spaces until the values have equal lengths; then they are compared for the full length.

No more than 195 columns of the same table can be VARCHAR data types.

Nonprintable Characters with VARCHAR

Nonprintable VARCHAR characters are entered, displayed, and treated in the same way that nonprintable characters in CHAR values are treated. For details, see "Nonprintable Characters with CHAR" on page 2-10.

Storing Numeric Values in a VARCHAR Column

When you insert a numeric value in a VARCHAR column, the stored value does not get padded with trailing blanks to the maximum length of the column. The number of digits in a numeric VARCHAR value is the number of characters that are required to store that value. For example, in the next example, the value stored in table **mytab** is 1.

```
create table mytab (col1 varchar(10));
insert into mytab values (1);
```

Tip: VARCHAR treats C *null* (binary 0) and string terminators as termination characters for nonprintable characters.

In some East Asian locales, VARCHAR data types can store multibyte characters if the database locale supports a multibyte code set. If you store multibyte characters, make sure to calculate the number of bytes needed. For more information, see the *IBM Informix GLS User's Guide*.

Multibyte Characters with VARCHAR

The first parameter in VARCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR feature that is described in the section "Logical Character Semantics in Character Type Declarations" on page 2-37.

Collating VARCHAR Values

The main difference between the NVARCHAR and the VARCHAR data types (like the difference between CHAR and NCHAR) is the difference in collating order. In general, collation of VARCHAR (like CHAR and LVARCHAR) values is in the order of the characters as they exist in the code set.

An exception is the MATCHES operator, which applies a localized collation to NVARCHAR and VARCHAR values (and to CHAR, LVARCHAR, and NCHAR values) if you use bracket ([ ]) symbols to define ranges when **DB_LOCALE** (or SET COLLATION) has specified a localized collating order. For more information, see the *IBM Informix GLS User's Guide*.

Built-In Data Types

IBM Informix supports the following built-in data types.

Category	Data Types
Character	CHAR, CHARACTER VARYING, LVARCHAR, NCHAR, NVARCHAR, VARCHAR, IDSSECURITYLABEL
Large-object	Simple-large-object types: BYTE, TEXT Smart-large-object types: BLOB, CLOB
Logical	BOOLEAN
Multirepresentational	BSON and JSON built-in opaque data types
Numeric	BIGINT, BIGSERIAL, DECIMAL, FLOAT, INT8, INTEGER, MONEY, SERIAL, SERIAL8, SMALLFLOAT, SMALLINT
Time	DATE, DATETIME, INTERVAL

Related reference:

"Summary of data types" on page 2-1

Related information:

BSON and JSON built-in opaque data types

Character Data Types

The character data types store string values.

Built-in Character Types

Table 2-8. Attributes of built-in character data types

	Size (in bytes)	Default	Reserved	Collation	Length
CHAR(n)	1 to 32,767	1 byte	None	Code set	Fixed
NCHAR(n)	1 to 32,767	1 byte	None	Localized	Fixed
VARCHAR(m, r)	1 to 255	0 for **r**	0 to 255 bytes	Code set	Variable
NVARCHAR(m, r)	1 to 255	0 for **r**	0 to 255 bytes	Localized	Variable
LVARCHAR(m)	1 to 32,739	2048 bytes	None	Code set	Variable

The database server also uses LVARCHAR to represent the external format of opaque data types. In I/O operations of the database server, LVARCHAR data values have no upper limit on their size, apart from file size restrictions or limits of your operating system or hardware resources.

Logical Character Semantics in Character Type Declarations

IBM Informix supports a configuration parameter, SQL_LOGICAL_CHAR, whose setting can instruct the SQL parser to interpret the maximum size of character columns in data type declarations of the CREATE TABLE or ALTER TABLE statements as logical characters, rather than in units of bytes.

When a database is created, the current SQL_LOGICAL_CHAR setting for the database server is recorded in the **systables** table of the system catalog. The feature has no effect on tables that are subsequently created or altered in the database if the setting is OFF or 1.

In a database where the SQL_LOGICAL_CHAR setting is `ON` or is a digit between 2, 3, or 4, however, the SQL parser interprets explicit and implicit size declarations as logical characters in declarations of SPL variables and declarations of columns in database tables for the following character types:

- CHAR and CHARACTER
- CHARACTER VARYING and VARCHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- DISTINCT types of the data types listed above
- DISTINCT types of those DISTINCT types
- ROW data type fields of the types listed above .
- LIST, MULTISET, and SET elements of the types listed above.

This feature has no effect on the maximum storage size limits for the character types listed in the previous table. For databases that use a multibyte locale, however, it can reduce the risk of data truncation when a string is inserted into a character column or assigned to a character variable.

For example, if 4 is the SQL_LOGICAL_CHAR setting for the database, then a VARCHAR(10, 5) specification is interpreted as requesting a maximum of 40 bytes of storage, with 5 of these bytes reserved, creating a VARCHAR(40, 5) data type in standard SQL notation, rather than what was specified in the declaration.

The reserve size parameters of VARCHAR and NVARCHAR data types are not affected by the SQL_LOGICAL_CHAR setting, because the minimum size of a multibyte character is 1 byte. In this example, the minimum size of 5 multibyte characters is 5 bytes, a size that remains unchanged.

See the description of the SQL_LOGICAL_CHAR configuration parameter in the *IBM Informix Administrator's Reference* for more information about the effect of the SQL_LOGICAL_CHAR setting in databases whose **DB_LOCALE** specifies a multibyte locale. For additional information about multibyte locales and logical characters, see the *IBM Informix GLS User's Guide*.

IDSSECURITYLABEL

IBM Informix also supports the IDSSECURITYLABEL data type for systems that implement label-based access control (LBAC). This built-in data type can be formally classified as a character type, because it is defined as a DISTINCT OF VARCHAR(128) data type, but only users who hold the **DBSECADM** role can declare this data type in DDL operations. It supports the LBAC security feature, rather than functioning as a general-purpose character type.

Data Type Promotion

For some string-manipulation operations of IBM Informix, the five built-in character data types listed above support data type promotion, in order to reduce the risk of string operations failing because a returned string is too large to be stored in an NVARCHAR or VARCHAR column or program variable. See the topic "Return Types from CONCAT and String Manipulation Functions" in *IBM Informix Guide to SQL: Syntax* for details of data type promotion among IBM Informix character types.

National Language Support

The NCHAR and NVARCHAR types are sometimes called National Language Support data types because of their support for localized collation. Because columns of type VARCHAR or NVARCHAR have no default size, you must specify a size (no greater than 255) in their declaration. For VARCHAR or NVARCHAR columns on which an index is defined, the maximum size is 254 bytes.

NLSCASE INSENSITIVE Databases

In databases created with the `NLSCASE INSENSITIVE` keyword option, operations on data strings of the NCHAR or NVARCHAR types makes no distinction between uppercase and lowercase variants of the same letter. Rows are stored in NCHAR or NVARCHAR columns in the letter case in which characters were loaded, but data strings that consist of the same letters in the same sequence are evaluated as duplicates, even if the case of some letters is not identical. For example, the three NCHAR strings `"ABC"` and `"AbC"` and `"abC"` are treated as duplicates. Other built-in character types, including CHAR, LVARCHAR, and VARCHAR, follow the default case-sensitive rules, so that the same three strings in a CHAR column evaluate to distinct values.

Databases with the `NLSCASE INSENSITIVE` property also ignore the letter case of DISTINCT data types whose base types are NCHAR or NVARCHAR, as well as NCHAR or NVARCHAR fields in named or unnamed ROW types, and NCHAR or NVARCHAR elements of COLLECTION data types, including LIST, SET, or MULTISET.

In a database that is insensitive to the letter case of NCHAR or NVARCHAR values, string manipulation operations might produce unexpected results, if they implicitly cast CHAR, LVARCHAR, or VARCHAR operands or arguments to NCHAR or NVARCHAR data types. In some contexts, the operation can return a duplicate string, despite letter case variations that the database server would not have treated as duplicates for the original data types.

Large-Object Data Types

A large object is a data object that is logically stored in a table column but physically stored independent of the column. Large objects are stored separate from the table because they typically store a large amount of data. Separation of this data from the table can increase performance.

Figure 2-3 shows the large-object data types.

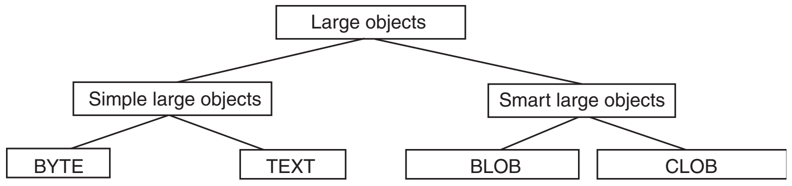


Figure 2-3. Large-Object Data Types

Only IBM Informix supports BLOB and CLOB data types.

For the relative advantages and disadvantages of simple and smart large objects, see the *IBM Informix Database Design and Implementation Guide*.

Simple Large Objects

Simple large objects are a category of large objects that have a theoretical size limit of 2^{31} bytes and a practical limit that your disk capacity determines. IBM Informix supports these simple-large-object data types:

BYTE Stores binary data. For more detailed information about this data type, see the description on page "BYTE data type" on page 2-8.

TEXT Stores text data. For more detailed information about this data type, see the description on page "TEXT data type" on page 2-33.

No more than 195 columns of the same table can be declared as BYTE or TEXT data types. Unlike smart large objects, simple large objects do not support random access to the data. When you transfer a simple large object between a client application and the database server, you must transfer the entire BYTE or TEXT value. If the data cannot fit into memory, you must store the data value in an operating-system file and then retrieve it from that file.

The database server stores simple large objects in *blobspaces*. A *blobspace* is a logical storage area that contains one or more chunks that only store BYTE and TEXT data. For information about how to define blobspaces, see your *IBM Informix Administrator's Guide*.

Smart large objects

Smart large objects are a category of large objects that support random access to the data, and that are generally recoverable.

The random access feature allows you to seek and read through the smart large object as if it were an operating-system file.

Smart large objects are also useful for opaque data types with large storage requirements. (See the description of opaque data types in "Opaque Data Types" on page 2-49.) They have a theoretical size limit of 2^{42} bytes and a practical limit that your disk capacity determines.

IBM Informix supports the following smart-large-object data types:

BLOB Stores binary data. For more information about this data type, see the description on page "BLOB data type" on page 2-7.

CLOB Stores text data. For more information about this data type, see "CLOB data type" on page 2-11.

IBM Informix stores smart large objects in *sbspaces*. An *sbspace* is a logical storage area that contains one or more chunks that store only BLOB and CLOB data. For information about how to define sbspaces, see your *IBM Informix Performance Guide*.

When you define a BLOB or CLOB column, you can determine the following large-object characteristics:

- LOG and NOLOG: whether the database server should log the smart large object in accordance with the current database logging mode.
- KEEP ACCESS TIME and NO KEEP ACCESS TIME: whether the database server should keep track of the last time the smart large object was accessed.
- HIGH INTEG and MODERATE INTEG: whether the database server should use sbspace page headers and page footers to detect data corruption (HIGH INTEG), or only use page headers (MODERATE INTEG).

Use of these characteristics can affect performance. For information, see your *IBM Informix Performance Guide*.

When an SQL statement accesses a smart-large-object, the database server does not send the actual BLOB or CLOB data. Instead, it establishes a pointer to the data and returns this pointer. The client application can then use this pointer in open, read, or write operations on the smart large object.

To access a BLOB or CLOB column from within a client application, use one of the following application programming interfaces (APIs):

- From within IBM Informix ESQL/C programs, use the smart-large-object API. (For more information, see the *IBM Informix ESQL/C Programmer's Manual*.)
- From within a DataBlade module, use the Client and Server API. (For more information, see the *IBM Informix DataBlade API Programmer's Guide*.)

For information about smart large objects, see the *IBM Informix Guide to SQL: Syntax* and *IBM Informix Database Design and Implementation Guide*.

Time Data Types

DATE and DATETIME data values represent zero-dimensional points in time; INTERVAL data values represent 1-dimensional spans of time, with positive or negative values. DATE precision is always an integer count of days, but various field qualifiers can define the DATETIME and INTERVAL precision. You can use DATE, DATETIME, and INTERVAL data in arithmetic and relational expressions. You can manipulate a DATETIME value with another DATETIME value, an INTERVAL value, the current time (specified by the keyword CURRENT), or some unit of time (using the keyword UNITS).

You can use a DATE value in most contexts where a DATETIME value is valid, and vice versa. You also can use an INTERVAL operand in arithmetic operations where a DATETIME value is valid. In addition, you can add two INTERVAL values and multiply or divide an INTERVAL value by a number.

An INTERVAL column can hold a value that represents the difference between two DATETIME values or the difference between (or sum of) two INTERVAL values. In either case, the result is a span of time, which is an INTERVAL value. Conversely, if you add or subtract an INTERVAL from a DATETIME value, another DATETIME value is produced, because the result is a specific time.

Table 2-9 lists the binary arithmetic operations that you can perform on DATE, DATETIME, and INTERVAL operands, and the data type that is returned by the arithmetic expression.

Table 2-9. Arithmetic Operations on DATE, DATETIME, and INTERVAL Values

Operand 1	Operator	Operand 2	Result
DATE	**-**	DATETIME	INTERVAL
DATETIME	**-**	DATE	INTERVAL
DATE	**+** *or* **-**	INTERVAL	DATETIME
DATETIME	**-**	DATETIME	INTERVAL
DATETIME	**+** *or* **-**	INTERVAL	DATETIME
INTERVAL	**+**	DATETIME	DATETIME
INTERVAL	**+** *or* **-**	INTERVAL	INTERVAL

Table 2-9. Arithmetic Operations on DATE, DATETIME, and INTERVAL Values (continued)

Operand 1	Operator	Operand 2	Result
DATETIME	-	CURRENT	INTERVAL
CURRENT	-	DATETIME	INTERVAL
INTERVAL	+	CURRENT	DATETIME
CURRENT	+ *or* -	INTERVAL	DATETIME
DATETIME	+ *or* -	UNITS	DATETIME
INTERVAL	+ *or* -	UNITS	INTERVAL
INTERVAL	* *or* /	NUMBER	INTERVAL

No other combinations are allowed. You cannot add two DATETIME values because this operation does not produce either a specific time or a span of time. For example, you cannot add December 25 and January 1, but you can subtract one from the other to find the time span between them.

Manipulating DATETIME Values

You can subtract most DATETIME values from each other.

Dates can be in any order and the result is either a positive or a negative INTERVAL value. The first DATETIME value determines the precision of the result, which includes the same time units as the first operand.

If the second DATETIME value has fewer fields than the first, the precision of the second operand is increased automatically to match the first.

In the following example, subtracting the DATETIME YEAR TO HOUR value from the DATETIME YEAR TO MINUTE value results in a positive interval value of 60 days, 1 hour, and 30 minutes. Because minutes were not included in the second operand, the database server sets the minutes value for the second operand to 0 before performing the subtraction.

```
DATETIME (2003-9-30 12:30) YEAR TO MINUTE
   - DATETIME (2003-8-1 11) YEAR TO HOUR

Result: INTERVAL (60 01:30) DAY TO MINUTE
```

If the second DATETIME operand has more fields than the first (regardless of whether the precision of the extra fields is larger or smaller than those in the first operand), the additional time unit fields in the second value are ignored in the calculation.

In the next expression (and its result), the year is not included for the second operand. Therefore, the year is set automatically to the current year (from the system clock-calendar), in this example 2005, and the resulting INTERVAL is negative, which indicates that the second date is later than the first.

```
DATETIME (2005-9-30) YEAR TO DAY
   - DATETIME (10-1) MONTH TO DAY

Result: INTERVAL (-1) DAY TO DAY [assuming that the current
                                  year is 2005]
```

You can compare two DATETIME values by using the **mi_datetime_compare()** function.

Related reference:

"DATETIME data type" on page 2-12

Related information:

The mi_datetime_compare() function

Manipulating DATETIME with INTERVAL Values

INTERVAL values can be added to or subtracted from DATETIME values. In either case, the result is a DATETIME value. If you are adding an INTERVAL value to a DATETIME value, the order of values is unimportant; however, if you are subtracting, the DATETIME value must come first. Adding or subtracting a positive INTERVAL value moves the DATETIME result forward or backward in time. The expression shown in the following example moves the date ahead by three years and five months:

```
DATETIME (2000-8-1) YEAR TO DAY
   + INTERVAL (3-5) YEAR TO MONTH

Result: DATETIME (2004-01-01) YEAR TO DAY
```

Important: Evaluate the logic of your addition or subtraction. Remember that months can have 28, 29, 30, or 31 days and that years can have 365 or 366 days.

In most situations, the database server automatically adjusts the calculation when the operands do not have the same precision. In certain contexts, however, you must explicitly adjust the precision of one value to perform the calculation. If the INTERVAL value you are adding or subtracting has fields that are not included in the DATETIME value, you must use the EXTEND function to increase the precision of the DATETIME value. (For more information about the EXTEND function, see the Expression segment in the *IBM Informix Guide to SQL: Syntax*.)

For example, you cannot subtract an INTERVAL MINUTE TO MINUTE value from the DATETIME value in the previous example that has a YEAR TO DAY field qualifier. You can, however, use the EXTEND function to perform this calculation, as the following example shows:

```
EXTEND (DATETIME (2008-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE(3) TO MINUTE

Result: DATETIME (2008-07-31 12:00) YEAR TO MINUTE
```

The EXTEND function allows you to explicitly increase the DATETIME precision from YEAR TO DAY to YEAR TO MINUTE. This allows the database server to perform the calculation, with the resulting extended precision of YEAR TO MINUTE.

Manipulating DATE with DATETIME and INTERVAL Values

You can use DATE operands in some arithmetic expressions with DATETIME or INTERVAL operands by writing expressions to do the manipulating, as Table 2-10 shows.

Table 2-10. Results of Expressions That Manipulate DATE with DATETIME or INTERVAL Values

Expression	**Result**
DATE – DATETIME	INTERVAL
DATETIME – DATE	INTERVAL
DATE + *or* – INTERVAL	DATETIME

In the cases that Table 2-10 on page 2-43 shows, DATE values are first converted to their corresponding DATETIME equivalents, and then the expression is evaluated by the rules of arithmetic.

Although you can interchange DATE and DATETIME values in many situations, you must indicate whether a value is a DATE or a DATETIME data type. A DATE value can come from the following sources:
- A column or program variable of type DATE
- The TODAY keyword
- The DATE( ) function
- The MDY function
- A DATE literal

A DATETIME value can come from the following sources:
- A column or program variable of type DATETIME
- The CURRENT keyword
- The EXTEND function
- A DATETIME literal

The database locale defines the default DATE and DATETIME formats. For the default locale, U.S. English, these formats are '*mm/dd/yy*' for DATE values and '*yyyy-mm-dd hh*:MM:*ss*' for DATETIME values.

To represent DATE and DATETIME values as character strings, the fields in the strings must be in the required order. In other words, when a DATE value is expected, the string must be in DATE format and when a DATETIME value is expected, the string must be in DATETIME format. For example, you can use the string `10/30/2008` as a DATE string but not as a DATETIME string. Instead, you must use `2008-10-30` or `08-10-30` as the DATETIME string.

In a nondefault locale, literal DATE and DATETIME strings must match the formats that the locale defines. For more information, see the *IBM Informix GLS User's Guide*.

You can customize the DATE format that the database server expects with the **DBDATE** and **GL_DATE** environment variables. You can customize the DATETIME format that the database server expects with the **DBTIME** and **GL_DATETIME** environment variables. For more information, see "DBDATE environment variable" on page 3-22 and "DBTIME environment variable" on page 3-32. For more information about all these environment variables, see the *IBM Informix GLS User's Guide*.

You can also subtract one DATE value from another DATE value, but the result is a positive or negative INTEGER count of days, rather than an INTERVAL value. If an INTERVAL value is required, you can either use the UNITS DAY operator to convert the INTEGER value into an INTERVAL DAY TO DAY value, or else use EXTEND to convert one of the DATE values into a DATETIME value before subtracting.

For example, the following expression uses the **DATE**( ) function to convert character string constants to DATE values, calculates their difference, and then uses the UNITS DAY keywords to convert the INTEGER result into an INTERVAL value:

```
(DATE ('5/2/2007') - DATE ('4/6/1968')) UNITS DAY

Result: INTERVAL (12810) DAY(5) TO DAY
```

Important: Because of the high precedence of UNITS relative to other SQL operators, you should generally enclose any arithmetic expression that is the operand of UNITS within parentheses, as in the preceding example.

If you need YEAR TO MONTH precision, you can use the EXTEND function on the first DATE operand, as the following example shows:

```
EXTEND (DATE ('5/2/2007'), YEAR TO MONTH) - DATE ('4/6/1969')

Result: INTERVAL (39-01) YEAR TO MONTH
```

The resulting INTERVAL precision is YEAR TO MONTH, because the DATETIME value came first. If the DATE value had come first, the resulting INTERVAL precision would have been DAY(5) TO DAY.

Related reference:

"DATETIME data type" on page 2-12

"INTERVAL data type" on page 2-19

Manipulating INTERVAL Values

You can add or subtract INTERVAL values only if both values are from the same class; that is, if both are year-month or both are day-time.

In the following example, a SECOND TO FRACTION value is subtracted from a MINUTE TO FRACTION value:

```
INTERVAL (100:30.0005) MINUTE(3) TO FRACTION(4)
   - INTERVAL (120.01) SECOND(3) TO FRACTION

Result: INTERVAL (98:29.9905) MINUTE TO FRACTION(4)
```

The use of numeric qualifiers alerts the database server that the MINUTE and FRACTION in the first value and the SECOND in the second value exceed the default number of digits.

When you add or subtract INTERVAL values, the second value cannot have a field with greater precision than the first. The second INTERVAL, however, can have a field of smaller precision than the first. For example, the second INTERVAL can be HOUR TO SECOND when the first is DAY TO HOUR. The additional fields (in this case MINUTE and SECOND) in the second INTERVAL value are ignored in the calculation.

You can compare two INTERVAL values by using the **mi_interval_compare()** function.

Related reference:

"INTERVAL data type" on page 2-19

Related information:

The mi_interval_compare() function

Multiplying or Dividing INTERVAL Values

You can multiply or divide INTERVAL values by numbers. Any remainder from the calculation is ignored, however, and the result is truncated to the precision of the INTERVAL. The following expression multiplies an INTERVAL value by a literal number that has a fractional part:

```
INTERVAL (15:30.0002) MINUTE TO FRACTION(4) * 2.5

Result: INTERVAL (38:45.0005) MINUTE TO FRACTION(4)
```

In this example, 15 * 2.5 = 37.5 minutes, 30 * 2.5 = 75 seconds, and 2 * 2.5 = 5 FRACTION (4). The 0.5 minute is converted into 30 seconds and 60 seconds are converted into 1 minute, which produces the final result of 38 minutes, 45 seconds, and 0.0005 of a second. The result of any calculation has the same precision as the original INTERVAL operand.

Extended Data Types

IBM Informix enables you to create *extended data types* to characterize data that cannot easily be represented with the built-in data types. (You cannot, however, use extended data types in distributed transactions that query external tables.) You can create these categories of extended data types:

- Complex data types
- Distinct data types
- Opaque data types

Sections that follow provide an overview of each of these data types.

For more information about extended data types, see the *IBM Informix Database Design and Implementation Guide* and *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Related reference:

"Summary of data types" on page 2-1

Complex data types

A *complex data type* can store one or more values of other built-in or extended data types.

Figure 2-4 shows the complex types that IBM Informix supports.

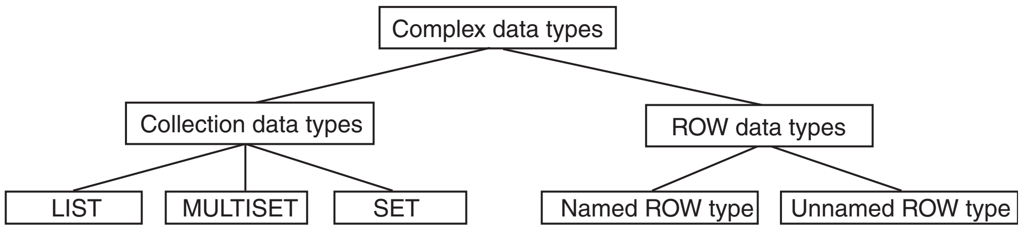


Figure 2-4. Complex Data Types of IBM Informix

The following table summarizes the structure of the complex data types.

Table 2-11. Collection types are complex data types that are made up of elements, each of which is of the same data type.

Collection types	Description
LIST	A group of ordered elements, each of which need not be unique within the group.
MULTISET	A group of elements, each of which need not be unique. The order of the elements is ignored.

Table 2-11. Collection types are complex data types that are made up of elements, each of which is of the same data type. (continued)

Collection types	Description
SET	A group of elements, each of which is unique. The order of the elements is ignored.

Table 2-12. ROW types are complex data types that are made up of fields.

ROW types	Description
Named ROW type	Row types that are identified by their name.
Unnamed ROW type	Row types that are identified by their structure.

Complex data types can be nested. For example, you can construct a ROW type whose fields include one or more sets, multisets, ROW types, and lists. Likewise, a collection type can have elements whose data type is a ROW type or a collection type.

Complex types that include opaque types inherit the following support functions.
- **input**
- **output**
- **send**
- **recv**
- **import**
- **export**
- **import_binary**
- **export_binary**
- **assign**
- **destroy**
- **LO_handles**
- **hash**
- **lessthan**
- **equal**
- **lessthan** (for ROW types only)

The topics that follow summarize the complex data types. For more information, see the *IBM Informix Database Design and Implementation Guide*.

Collection Data Types

A collection data type is a complex type that is made up of one or more elements, all of the same data type. A collection element can be of any data type (including other complex types) except BYTE, TEXT, SERIAL, SERIAL8, or BIGSERIAL.

Important: An element cannot have a *NULL* value. You must specify the *NOT NULL* constraint for collection elements. No other constraints are valid for collections.

IBM Informix supports three kinds of built-in collection types: LIST, SET, and MULTISET. The keywords used to declare these collections are the names of the *type constructors* or just *constructors*. For the syntax of collection types, see the *IBM Informix Guide to SQL: Syntax*. No more than 97 columns of the same table can be declared as collection data types.

When you specify element values for a collection, list the element values after the constructor and between braces ({ }). For example, suppose you have a collection column with the following MULTISET data type:

```
CREATE TABLE table1
(
   mset_col MULTISET(INTEGER NOT NULL)
)
```

The next INSERT statement adds one group of element values to this column. (The word MULTISET in these two examples is the MULTISET constructor.)

```
INSERT INTO table1 VALUES (MULTISET{5, 9, 7, 5})
```

You can leave the braces empty to indicate an empty set:

```
INSERT INTO table1 VALUE (MULTISET{})
```

An empty collection is not equivalent to a NULL value for the column.

Accessing collection data:
To access the elements of a collection column, you must fetch the collection into a collection variable and modify the contents of the collection variable. Collection variables can be either of the following types:

- Variables in an SPL routine

  For more information, see the *IBM Informix Guide to SQL: Tutorial*.
- Host variables in IBM Informix ESQL/C programs

  For more information, see the *IBM Informix ESQL/C Programmer's Manual*.

You can also use nested dot notation to access collection data. For more about accessing elements of a collection, see the *IBM Informix Guide to SQL: Tutorial*.

Important: Collection data types are not valid as arguments to functions that are used for functional indexes.

ROW Data Types

A ROW data type is an ordered collection of one or more elements, called *fields*. Each field has a name and a data type. The fields of a ROW are comparable to the columns of a table, but with important differences:

- A field has no default clause.
- You cannot define constraints on a field.
- You can only use fields with row types, not with tables.

Two kinds of ROW data types exist:

- *Named ROW data types* are identified by their names.
- *Unnamed ROW data types* are identified by their structure.

The *structure* of an unnamed ROW data type is the number (and the order of data types) of its fields.

No more than 195 columns of the same table can be declared as ROW data types. For more information about ROW data types, see "ROW data type, Named" on page 2-27 and "ROW data type, Unnamed" on page 2-28.

You can cast between named and unnamed ROW data types; this is described in the *IBM Informix Database Design and Implementation Guide*.

Distinct Data Types

A distinct data type has the same internal structure as some other source data type in the database. The source type can be a built-in or extended data type. What distinguishes a distinct type from its source type are support functions that are defined on the distinct type.

No more than approximately 97 columns of the same table can be DISTINCT of collection data types (SET, LIST, and MULTISET). No more than approximately 195 columns of the same table can be DISTINCT types that are based on BYTE, TEXT, ROW, LVARCHAR, NVARCHAR, or VARCHAR source types. (Here 195 columns is an approximate lower limit that applies to platforms with a 2 Kb base page size. For platforms with a base page size of 4 Kb, such as Windows and AIX® systems, the upper limit is approximately 450 columns of these data types.) For more information, see the section "DISTINCT data types" on page 2-17. See also *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Opaque Data Types

An opaque data type is a user-defined or built-in data type that is fully encapsulated. The internal structure of an opaque data type is unknown to the database server.

Except for user-defined types (UDTs) that are DISTINCT of built-in non-opaque types, UDTs whose source types are built-in types are opaque data types. Similarly, UDTs that are DISTINCT of built-in opaque types are opaque types.

Built-in opaque data types

The built-in data types BLOB, BOOLEAN, CLOB, BSON, JSON, and LVARCHAR are implemented as opaque data types. You can access all of these in other databases of the same Informix instance, but you cannot access the BLOB or CLOB built-in opaque data types in cross-server distributed operations.

UDTs that are DISTINCT of built-in opaque types and that are cast to built-in types are valid in cross-server queries and other DML operations, but all the casts and all the DISTINCT OF definitions for the UDTs must be identical in every participating database.

Several system catalog tables, whose schema are shown in "Structure of the System Catalog" on page 1-7, have columns of built-in opaque data types. For information on how the system catalog emcodes columns of built-in opaque data types, see "SYSCOLUMNS" on page 1-17.

User-defined opaque data types

You must provide the following information to the database server for an opaque data type:

- A data structure for how the data values are stored on disk
- Support functions to determine how to convert between the disk storage format and the user format for data entry and display
- Secondary access methods that determine how the index on this data type is built, used, and manipulated

- User functions that use the data type
- A system catalog entry to register the opaque type in the database

The internal structure of an opaque type is not visible to the database server and can only be accessed through user-defined routines. Definitions for opaque types are stored in the sysxtdtypes system catalog table. These SQL statements maintain the definitions of opaque types in the database:

- The CREATE OPAQUE TYPE statement registers a new opaque type in the database.
- The DROP TYPE statement removes a previously defined opaque type from the database.

For more information, see the section "OPAQUE data types" on page 2-26. See also *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Data Type Casting and Conversion

Occasionally, the data type that was assigned to a column with the CREATE TABLE statement is inappropriate. You can change the data type of a column when you are required to store larger values than the current data type can accommodate. The database server allows you to change the data type of the column or to cast its values to a different data type with either of the following methods:

- Use the ALTER TABLE statement to modify the data type of a column.

  For example, if you create a SMALLINT column and later find that you must store integers larger than 32,767, you must change the data type of that column to store the larger value. You can use ALTER TABLE to change the data type to INTEGER. The conversion changes the data type of all values that currently exist in the column and any new values that might be added.
- Use the CAST AS keywords or the double colon (::) cast operator to cast a value to a different data type.

  Casting does not permanently alter the data type of a value; it expresses the value in a more convenient form. Casting user-defined data types into built-in types allows client programs to manipulate data types without knowledge of their internal structure.

If you change data types, the new data type must be able to store all of the old value.

Both data-type conversion and casting depend on casts registered in the **syscasts** system catalog table. For information about **syscasts**, see "SYSCASTS" on page 1-14.

A cast is either built-in or user defined. Guidelines exist for casting distinct and extended data types. For more information about casting opaque data types, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information about casting other extended data types see, the *IBM Informix Database Design and Implementation Guide*.

Using Built-in Casts

User **informix** owns built-in casts. They govern conversions from one built-in data type to another. Built-in casts allow the database server to attempt the following data-type conversions:

- A character type to any other character type
- A character type to or from another built-in type
- A numeric type to any other numeric type

The database server automatically invokes appropriate built-in casts when required. For time data types, conversion between DATE and DATETIME data types requires explicit casts with the EXTEND function, and explicit casts with the UNITS operator are required for number-to-INTERVAL conversion. Built-in casts are unavailable for converting large (BYTE, BLOB, CLOB, and TEXT) built-in types to other built-in data types.

When you convert a column from one built-in data type to another, the database server applies the appropriate built-in casts to each value already in the column. If the new data type cannot store any of the resulting values, the ALTER TABLE statement fails.

For example, if you try to convert a column from the INTEGER data type to the SMALLINT data type and the following values exist in the INTEGER column, the database server does not change the data type, because SMALLINT columns cannot accommodate numbers greater than 32,767:

```
100    400    700    50000      700
```

The same situation might occur if you attempt to transfer data from FLOAT or SMALLFLOAT columns to INTEGER, SMALLINT, or DECIMAL columns. Errors of overflow, underflow, or truncation can occur during data type conversion.

Sections that follow describe database server behavior during certain types of casts and conversions.

Converting from number to number

When you convert data from one number data type to another, you occasionally find rounding errors.

The following table indicates which numeric data type conversions are acceptable and what kinds of errors you can encounter when you convert between certain numeric data types. In the table, the following codes are used:

OK	No error
P	An error can occur, depending on the precision of the decimal
E	An error can occur, depending on the data value
D	No error, but less significant digits might be lost

Table 2-13. Acceptable conversions and possible errors

Target Type	SMALL INT	INTEGER	INT8	SMALL FLOAT	FLOAT	DECIMAL
SMALLINT	OK	OK	OK	OK	OK	OK
INTEGER	E	OK	OK	E	OK	P
INT8	E	E	OK	D	E	P
SMALLFLOAT	E	E	E	OK	OK	P
FLOAT	E	E	E	D	OK	P
DECIMAL	E	E	E	D	D	P

For example, if you convert a FLOAT value to DECIMAL(4,2), your database server rounds off the floating-point number before storing it as DECIMAL.

This conversion can result in an error depending on the precision assigned to the DECIMAL column.

Converting Between Number and Character

You can convert a character column (of a data type such as CHAR, NCHAR, NVARCHAR, or VARCHAR) to a numeric column. If a data string, however, contains any characters that are not valid in a number column (for example, the letter *l* instead of the number *1*), the database server returns an error.

You can also convert a numeric column to a character column. If the character column is not large enough to receive the number, however, the database server generates an error. If the database server generates an error, it cannot complete the ALTER TABLE statement or cast, and leaves the column values as characters. You receive an error message and the statement is rolled back automatically (regardless of whether you are in a transaction).

Converting Between INTEGER and DATE

You can convert an integer column (SMALLINT, INTEGER, or INT8) to a DATE value. The database server interprets the integer as a value in the internal format of the DATE column. You can also convert a DATE column to an integer column. The database server stores the internal format of the DATE column as an integer representing a Julian date.

Converting Between DATE and DATETIME

You can convert DATE columns to DATETIME columns. If the DATETIME column contains more fields than the DATE column, however, the database server either ignores the fields or fills them with zeros. The illustrations in the following list show how these two data types are converted (assuming that the default date format is *mm/dd/yyyy*):

- If you convert DATE to DATETIME YEAR TO DAY, the database server converts the existing DATE values to DATETIME values. For example, the value 08/15/2002 becomes 2002-08-15.
- If you convert DATETIME YEAR TO DAY to the DATE format, the value 2002-08-15 becomes 08/15/2002.
- If you convert DATE to DATETIME YEAR TO SECOND, the database server converts existing DATE values to DATETIME values and fills in the additional DATETIME fields with zeros. For example, 08/15/2002 becomes 2002-08-15 00:00:00.
- If you convert DATETIME YEAR TO SECOND to DATE, the database server converts existing DATETIME to DATE values but drops fields for time units smaller than DAY. For example, 2002-08-15 12:15:37 becomes 08/15/2002.

Using User-Defined Casts

Implicit and explicit casts are owned by the users who create them. They govern casts and conversions between user-defined data types and other data types. Developers of user-defined data types must create certain implicit and explicit casts and the functions that are used to implement them. The casts allow user-defined types to be expressed in a form that clients can manipulate.

For information about how to register and use implicit and explicit casts, see the CREATE CAST statement in the *IBM Informix Guide to SQL: Syntax* and the *IBM Informix Database Design and Implementation Guide*.

Implicit Casts

Implicit casts allow you to convert a user-defined data type to a built-in type or vice versa. The database server automatically invokes a single implicit cast when it must evaluate and compare expressions or pass arguments. Operations that require more than one implicit cast fail.

Users can explicitly invoke an implicit cast using the CAST AS keywords or the double colon (**::**) cast operator.

Explicit Casts

Explicit casts, unlike implicit casts or built-in casts, are *never* invoked automatically by the database server. Users must invoke them explicitly with the CAST AS keywords or with the double colon (**::**) cast operator.

Determining Which Cast to Apply

The database server uses the following rules to determine which cast to apply in a particular situation:

- To compare two built-in types, the database server automatically invokes the appropriate built-in casts.
- The database server applies only one implicit cast per operand. If two or more casts are required to convert the operand to the specified type, the user must explicitly invoke the additional casts.

  In the following example, the literal value `5.55` is implicitly cast to DECIMAL, and is then explicitly cast to MONEY, and finally to `yen`:

  ```
  CREATE DISTINCT TYPE yen AS MONEY
  . . .
  INSERT INTO currency_tab
     VALUES (5.55::MONEY::yen)
  ```
- To compare a distinct type to its source type, the user must explicitly cast one type to the other.
- To compare a distinct type to a type other than its source, the database server looks for an implicit cast between the source type and the specified type.

  If neither cast is registered, the user must invoke an explicit cast between the distinct type and the specified type. If this cast is not registered, the database server automatically invokes a cast from the source type to the specified type.

  If none of these casts is defined, the comparison fails.
- To compare an opaque type to a built-in type, the user must explicitly cast the opaque type to a data type that the database server understands (such as LVARCHAR, SENDRECV, IMPEXP, or IMPEXPBIN). The database server then invokes built-in casts to convert the results to the specified built-in type.
- To compare two opaque types, the user must explicitly cast one opaque type to a form that the database server understands (such as LVARCHAR, SENDRECV, IMPEXP, or IMPEXPBIN) and then explicitly cast this type to the second opaque type.

For information about casting and the BOOLEAN, BSON, JSON, IMPEXP, IMPEXPBIN, LVARCHAR, and SENDRECV built-in opaque data types, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Casts for distinct types

You define a distinct type based on a built-in type or an existing opaque type or ROW type. Although data of the distinct type has the same length and alignment and is passed in the same way as data of the source type, the two cannot be compared directly. To compare a distinct type and its source type, you must explicitly cast one type to the other.

When you create a new distinct type, the database server automatically registers two explicit casts:

- A cast from the distinct type to its source type
- A cast from the source type to the distinct type

You can create an implicit cast between a distinct type and its source type. To create an implicit cast, however, you must first drop the default explicit cast between the distinct type and its source type.

You also can use all casts that have been registered for the source type without modification on the distinct type. You can also create and register new casts and support functions that apply only to the distinct type.

For examples that show how to create a cast function for a distinct type and register the function as cast, see the *IBM Informix Database Design and Implementation Guide*.

Important: For releases of IBM Informix earlier than Version 9.21, distinct data types inherited the built-in casts that are provided for the source type. The built-in casts of the source type are not inherited by distinct data types in this release.

What Extended Data Types Can Be Cast?

The next table shows the extended data type combinations that you can cast.

Table 2-14. Extended data type combinations

Target Type	Opaque Type	Distinct Type	Named ROW Type	Unnamed ROW Type	Collection Type	Built-in Type
Opaque Type	Explicit or implicit	Explicit	Explicit	Not Valid	Not Valid	Explicit or implicit[3]
Distinct Type	Explicit[3]	Explicit	Explicit	Not Valid	Not Valid	Explicit or implicit
Named ROW Type	Explicit[3]	Explicit	Explicit[3]	Explicit[1]	Not Valid	Not Valid
Unnamed ROW Type	Not Valid	Not Valid	Explicit[1]	Implicit[1]	Not Valid	Not Valid
Collection Type	Not Valid	Not Valid	Not Valid	Not Valid	Explicit[2]	Not Valid
Built-in Type	Explicit or implicit[3]	Explicit or implicit	Not Valid	Not Valid	Not Valid	System defined (implicit)

[1] Applies when two ROW types are structurally equivalent or casts exist to handle data conversions where corresponding field types are not the same.

[2] Applies when a cast exists to convert between the element types of the respective collection types.

[3] Applies when a user-defined cast exists to convert between the two data types.

The table shows only whether a cast between a source type and a target type are possible. In some cases, you must first create a user-defined cast before you can perform a conversion between two data types. In other cases, the database server provides either an implicit cast or a built-in cast that you must explicitly invoke.

Operator Precedence

An *operator* is a symbol or keyword that can be in an SQL expression. Most SQL operators are restricted in the data types of their operands and returned values. Some operators only support operands of built-in data types; others can support built-in and extended data types as operands.

The following table shows the precedence of the operators thatIBM Informix supports, in descending (highest to lowest) order of precedence. Operators with the same precedence are listed in the same row.

Operator Precedence	Example in Expression
. (*membership*) **[ ]** (*substring*)	**customer.phone** [1, 3]
UNITS	**x** UNITS DAY
+ - (*unary*)	**- y**
:: (*cast*)	NULL::TEXT
*** /**	**x / y**
+ - (*binary*)	**x -y**
|| (*concatenation*)	**customer.fname || customer.lname**
ANY ALL SOME	**orders.ship_date** > SOME (SELECT **paid_date** FROM **orders**)
NOT	NOT **y**
< <= = > >= != <>	**x >= y**
IN BETWEEN ... AND LIKE MATCHES	**customer.fname** MATCHES **y**
AND	**x** AND **y**
OR	**x** OR **y**

See the *IBM Informix Guide to SQL: Syntax* for the syntax and semantics of these SQL operators.

Chapter 3. Environment variables

Various *environment variables* affect the functionality of your IBM Informix products. You can set environment variables that identify your terminal, specify the location of your software and define other parameters.

Some environment variables are required; others are optional. You must either set or accept the default setting for required environment variables.

These topics describe how to use the environment variables that apply to one or more IBM Informix products and shows how to set them.

Types of environment variables

Two types of environment variables are explained in this chapter:

- Environment variables that are specific to IBM Informix

  Set IBM Informix environment variables when you want to work with IBM Informix products. Each IBM Informix product publication specifies the environment variables that you must set to use that product.
- Environment variables that are used with a specific operating system

  IBM Informix products rely on the correct setting of certain standard operating system environment variables. For example, you must always set the **PATH** environment variable.

In a UNIX environment, you might also be required to set the **TERMCAP** or **TERMINFO** environment variable to use some products effectively.

The GLS environment variables that support nondefault locales are described in the *IBM Informix GLS User's Guide*. The GLS variables are included in the list of environment variables in Table 3-1 on page 3-9.

The database server uses the environment variables that were in effect at the time when the database server was initialized.

The **onstat - g env** command lists the active environment settings.

Tip: Additional environment variables that are specific to your client application or SQL API might be explained in the publication for that product.

Important: Do not set any environment variable in the home directory of user **informix** (or in the file `.informix` in that directory) while initializing the database and creating the **sysmaster** database.

Limitations on environment variables

Size of a block of environment variables

At the start of a session, the client groups all the environment variables that the server will use and sends the environment variables to the server as single block. The maximum size of this block is 32K. If the block of environment variables is greater than 32K, the error -1832 is returned to the application. The text of this error is "Environment block is greater than 32K."

To resolve this error, you can either unset one or more environment variables or reduce the size of some of the environment variables.

Using environment variables on UNIX

You can set, unset, modify, and view environment variables. If you already use any IBM Informix products, some or all of the appropriate environment variables might be set.

You can set environment variables on UNIX in the following places:

- At the system prompt on the command line

  When you set an environment variable at the system prompt, you must reassign it the next time you log in to the system.
- In an environment-configuration file

  An environment-configuration file is a common or private file where you can set all the environment variables that IBM Informix products use. The use of such files reduces the number of environment variables that you must set at the command line or in a shell file.
- In a login file

  Values of environment variables set in your `.login`, `.cshrc`, or `.profile` file are assigned automatically every time you log in to the system.
- In the SET ENVIRONMENT statement of SQL

  Values of some environment variables can reset by the SET ENVIRONMENT statement. The scope of the new settings is generally the routine that executed the SET ENVIRONMENT statement, but it is the current session for the **OPTCOMPIND** environment variable of Informix, as described in the section "OPTCOMPIND environment variable" on page 3-61. For more information about these routines and on the SET ENVIRONMENT statement, see the *IBM Informix Guide to SQL: Syntax*.

In IBM Informix ESQL/C, you can set supported environment variables within an application with the **putenv()** system call and retrieve values with the **getenv()** system call, if your UNIX system supports these functions. For more information about **putenv()** and **getenv()**, see the *IBM Informix ESQL/C Programmer's Manual* and your C documentation.

Setting environment variables in a configuration file

The common (shared) environment-configuration file that is provided with IBM Informix products is located in **$INFORMIXDIR/etc/informix.rc**. Permissions for this shared file must be set to `644`.

A user can override the system or shared environment variables by setting variables in a private environment-configuration file. This file must have all of the following characteristics:

- Stored in the user's home directory
- Named **.informix**
- Permissions set to readable by the user

An environment-configuration file can contain comment lines (preceded by the # comment indicator) and variable definition lines that set values (separated by blank spaces or tabs), as the following example shows:

```
# This is an example of an environment-configuration file
#
DBDATE DMY4-
#
# These are ESQL/C environment variable settings
#
INFORMIXC gcc
CPFIRST TRUE
```

You can use the **ENVIGNORE** environment variable, described in "ENVIGNORE environment variable (UNIX)" on page 3-38, to override one or more entries in an environment-configuration file. Use the IBM Informix **chkenv** utility, described in "Checking environment variables with the chkenv utility" on page 3-4, to perform a sanity check on the contents of an environment-configuration file. The **chkenv** utility returns an error message if the file contains a bad environment variable or if the file is too large.

The first time you set an environment variable in a shell file or environment-configuration file, you must tell the shell process to read your entry before you work with your IBM Informix product. If you use a C shell, **source** the file; if you use a Bourne or Korn shell, use a period (**.**) to execute the file.

Setting environment variables at login time

Add commands that set your environment variables to the appropriate login file:

For C shell
.login or **.cshrc**

For Bourne shell or Korn shell
.profile

Syntax for setting environment variables

Use standard UNIX commands to set environment variables. The examples in the following table show how to set the ABCD environment variable to *value* for the C shell, Bourne shell, and Korn shell. The Korn shell also supports a shortcut, as the last row indicates. Environment variables are case-sensitive.

Shell	Command
C	`setenv ABCD value`

Bourne	`ABCD=value`
	`export ABCD`

Korn	`ABCD=value`
	`export ABCD`

Korn	`export ABCD=value`

The following diagram shows how the syntax for setting an environment variable is represented throughout this chapter. These diagrams indicate the setting for the C shell; for the Bourne or Korn shells, use the syntax illustrated in the preceding table.

```
►►—setenv—ABCD—value———————————————————————————————————————————►◄
```

Unsetting environment variables

To unset an environment variable, enter the following command.

Shell	Command
C	`unsetenv ABCD`
Bourne or Korn	`unset ABCD`

Modifying an environment-variable setting

Sometimes you must add information to an environment variable that is already set. For example, the **PATH** environment variable is always set on UNIX. When you use IBM Informix productd, you must add to the **PATH** setting the name of the directory where the executable files for the IBM Informix products are stored.

In the following example, the **INFORMIXDIR** is **/usr/informix**. (That is, during installation, the IBM Informix products were installed in the **/usr /informix** directory.) The executable files are in the **bin** subdirectory, **/usr/informix/bin**. To add this directory to the front of the C shell **PATH** environment variable, use the following command:

```
setenv PATH /usr/informix/bin:$PATH
```

Rather than entering an explicit pathname, you can use the value of the **INFORMIXDIR** environment variable (represented as **$INFORMIXDIR**), as the following example shows:

```
setenv INFORMIXDIR /usr/informix
setenv PATH $INFORMIXDIR/bin:$PATH
```

You might prefer to use this version to ensure that your **PATH** entry does not conflict with the search path that was set in **INFORMIXDIR**, and so that you are not required to reset **PATH** whenever you change **INFORMIXDIR**. If you set the **PATH** environment variable on the C shell command line, you might be required to include braces ({}) with the existing **INFORMIXDIR** and **PATH**, as the following command shows:

```
setenv PATH ${INFORMIXDIR}/bin:${PATH}
```

For more information about how to set and modify environment variables, see the publications for your operating system.

Viewing your environment-variable settings

After you install one or more IBM Informix products, enter the following command at the system prompt to view your current environment settings.

UNIX version	Command
BSD UNIX	**env**
UNIX System V	**printenv**

Checking environment variables with the chkenv utility

The **chkenv** utility checks the validity of shared or private environment-configuration files. It validates the names of the environment variables in the file, but not their values. Use **chkenv** to provide debugging information when you define, in an environment-configuration file, all the environment variables that your IBM Informix products use.

```
►►──chkenv──┬──────────┬──filename──────────────────────────────────────►◄
            └─pathname─┘
```

filename
: is the name of the environment-configuration file to be debugged.

pathname
: is the full directory path in which the environment variable file is located.

File **$INFORMIXDIR/etc/informix.rc** is the shared environment-configuration file. A private environment-configuration file is stored as **.informix** in the home directory of the user. If you specify no *pathname* for **chkenv**, the utility checks both the shared and private environment configuration files. If you provide a pathname, **chkenv** checks only the specified file.

Issue the following command to check the contents of the shared environment-configuration file:

```
chkenv informix.rc
```

The **chkenv** utility returns an error message if it finds a bad environment-variable name in the file or if the file is too large. You can modify the file and rerun the utility to check the modified environment-variable names.

IBM Informix products ignore all lines in the environment-configuration file, starting at the point of the error, if the **chkenv** utility returns the following message:

```
-33523     filename: Bad environment variable on line number.
```

If you want the product to ignore specified environment-variables in the file, you can also set the **ENVIGNORE** environment variable. For a discussion of the use and format of environment-configuration files and the **ENVIGNORE** environment variable, see page "ENVIGNORE environment variable (UNIX)" on page 3-38.

Rules of precedence for environment variables

When IBM Informix products accesses an environment variable, normally the following rules of precedence apply:

1. Of highest precedence is the value that is defined in the environment (shell) by explicitly setting the value at the shell prompt.
2. The second highest precedence goes to the value that is defined in the private environment-configuration file in the home directory of the user (**~/.informix**).
3. The next highest precedence goes to the value that is defined in the common environment-configuration file ($**INFORMIXDIR/etc/informix.rc**).
4. The lowest precedence goes to the default value, if one exists.

For precedence information about GLS environment variables, see the *IBM Informix GLS User's Guide*.

Important: If you set one or more environment variables before you start the database server, and you do not explicitly set the same environment variables for your client products, the clients will adopt the original settings.

Using environment variables on Windows

The following sections discuss setting, viewing, unsetting, and modifying environment variables for Windows applications.

Where to set environment variables on Windows

You can set environment variables in several places on Windows, depending on which IBM Informix application you use.

Environment variables can be set in several ways, as described in "Setting environment variables on Windows."

The SET ENVIRONMENT statement of SQL can set certain routine-specific environment options. For more information, see the description of SET ENVIRONMENT in the *IBM Informix Guide to SQL: Syntax*.

To use client applications such as IBM Informix ESQL/C or the Schema Tools on Windows environment, use the **Setnet32** utility to set environment variables. For information about the **Setnet32** utility, see the *IBM Informix Client Products Installation Guide* for your operating system.

In Informix ESQL/C, you can set supported environment variables within an application with the **ifx_putenv()** function and retrieve values with the **ifx_getenv()** function, if your Windows system supports them. For more information about **ifx_putenv()** and **ifx_getenv()**, see the *IBM Informix ESQL/C Programmer's Manual*.

Setting environment variables on Windows

You can set environment variables for command-prompt utilities in the following ways:

- With the System applet in the Control Panel
- In a command-line session

Using the system applet to change environment variables

The System applet provides a graphical interface to create, modify, and delete system-wide and user-specific variables. Environment variables that are set with the System applet are visible to all command-prompt sessions.

To change environment variables with the System applet in the control panel

1. Double-click the System applet icon from the Control Panel window.
2. Click the Environment tab near the top of the window.

   Two list boxes display System Environment Variables and User Environment Variables. System Environment Variables apply to an entire system, and User Environment Variables apply only to the sessions of the individual user.
3. To change the value of an existing variable, select that variable. The name of the variable and its current value are in the boxes at the bottom of the window.
4. To add a new variable, highlight an existing variable and type the new variable name in the box at the bottom of the window.
5. Next, enter the value for the new variable at the bottom of the window and click **Set** .
6. To delete a variable, select the variable and click **Delete**.

Important: In order to use the System applet to change System environment variables, you must belong to the Administrators group. For information about assigning users to groups, see your operating-system documentation.

Using the command prompt to change environment variables

You can change the setting of an environment variable at a command prompt.

The following diagram shows the syntax for setting an environment variable at a command prompt in Windows.

```
►►──set──ABCD──=──value───────────────────────────────────────────────►◄
```

If no *value* is specified, the environment variable is unset, as if it did not exist.

To view your current settings after one or more IBM Informix products are installed, enter the following command at the command prompt.

```
►►──set───────────────────────────────────────────────────────────────►◄
```

Sometimes you must add information to an environment variable that is already set. For example, the **PATH** environment variable is always set in Windows environments. When you use IBM Informix products, you must add the name of the directory where the executable files for the IBM Informix products are stored to the **PATH**.

In the following example, **INFORMIXDIR** is `d:\informix` (that is, during installation, IBM Informix products were installed in the `d: \informix` directory). The executable files are in the `bin` subdirectory, `d:\informix\bin`. To add this directory at the beginning of the **PATH** environment-variable value, use the following command:

```
set PATH=d:\informix\bin;%PATH%
```

Rather than entering an explicit pathname, you can use the value of the **INFORMIXDIR** environment variable (represented as **%INFORMIXDIR%**), as the following example shows:

```
set INFORMIXDIR=d:\informix
set PATH=%PATH%
```

You might prefer to use this version to ensure that your **PATH** entry does not contradict the search path that was set in **INFORMIXDIR** and to avoid the requirement to reset **PATH** whenever you change **INFORMIXDIR**.

For more information about setting and modifying environment variables, see your operating-system publications.

Using dbservername.cmd to initialize a command-prompt environment

Each time that you open a Windows command prompt, it acts as an independent environment. Therefore, environment variables that you set within it are valid only for that particular command-prompt instance.

For example, if you open one command window and set the variable, **`INFORMIXDIR`**, and then open another command window and type `set` to check your environment, you will find that **`INFORMIXDIR`** is not set in the new command-prompt session.

The database server installation program creates a batch file that you can use to configure command-prompt utilities, ensuring that your command-prompt environment is initialized correctly each time that you run a command-prompt session. The batch file, `dbservername.cmd`, is located in `%INFORMIXDIR%`, and is a plain text file that you can modify with any text editor. If you have more than one database server installed in `%INFORMIXDIR%`, there will be more than one batch file with the `.cmd` extension, each bearing the name of the database server with which it is associated.

To run `dbservername.cmd` from a command prompt, type `dbservername` or configure a command prompt so that it runs `dbservername.cmd` automatically at start.

Rules of precedence for Windows environment variables

When IBM Informix products access an environment variable, normally the following rules of precedence apply:

1. The setting in Setnet32 with the **Use my settings** box selected.
2. The setting in Setnet32 with the **Use my settings** box cleared.
3. The setting on the command line before running the application.
4. The setting in Windows as a user variable.
5. The setting in Windows as a system variable.
6. The lowest precedence goes to the default value.

An application examines the first five values as it starts. Unless otherwise stated, changing an environment variable after the application is running does not have any effect.

Environment variables in Informix products

The topics that follow discuss (in alphabetic order) environment variables that IBM Informix database server products and their utilities use.

Important: The descriptions of the following environment variables include the syntax for setting the environment variable on UNIX. For a general description of how to set these environment variables on Windows, see "Setting environment variables on Windows" on page 3-6.

Related information:

Informix environment variables with the IBM Informix JDBC Driver

GLS-related environment variables

Enterprise Replication configuration parameter and environment variable reference

AC_CONFIG file environment variable

Environment variable portal

This portal is an index of usage categories for IBM Informix and UNIX environment variables. The portal contains links to the topics that describe the environment variables.

Because the following table contains a comprehensive list of categories with links to applicable topics. Some environment variables are applicable for more than one category.

Table 3-1. Uses for environment variables

Functional category	Environment variable
Abbreviated year values	Specify how to expand literal DATE and DATETIME values: "DBCENTURY environment variable" on page 3-20
ANSI/ISO SQL compliance	Set the case of owner names: "ANSIOWNER environment variable" on page 3-16 Specify if you want to check for IBM Informix extensions to ANSI-standard SQL syntax: "DBANSIWARN environment variable" on page 3-19 No default table or routine access privileges for PUBLIC in databases not created WITH LOG MODE ANSI: "NODEFDAC environment variable" on page 3-59
archecker utility	Specify the full path name for the **archecker** configuration file: AC_CONFIG file environment variable
Buffers	Manage the fetch buffer size: "FET_BUF_SIZE environment variable" on page 3-39 Manage the network size: "IFX_NETBUF_SIZE environment variable" on page 3-44 Manage the network pool size: "IFX_NETBUF_PVTPOOL_SIZE environment variable (UNIX)" on page 3-44 Manage the BYTE or TEXT data buffer: "DBBLOBBUF environment variable" on page 3-20
Cache	Control the use of the shared-statement cache on a session: "STMT_CACHE environment variable" on page 3-71
Client/server	Specify the default database server: "INFORMIXSERVER environment variable" on page 3-54 Specify where shared-memory segments are attached to the client process: "INFORMIXSHMBASE environment variable (UNIX)" on page 3-55 Specify the stack size for a client process: "INFORMIXSTACKSIZE environment variable" on page 3-56 Specify locale information, including for the client and server: GLS-related environment variables
Code-set conversion	Specify locale and multibyte information: GLS-related environment variables Specify the location of the **concsm.cfg** file: "INFORMIXCONCSMCFG environment variable" on page 3-51 Specify the filename or pathname of the C compiler: "INFORMIXC environment variable (UNIX)" on page 3-50 Specify the pathname of the map file for C++ programs: "INFORMIXCPPMAP environment variable" on page 3-54 Specify information for compiling multithreaded IBM Informix ESQL/C applications: "THREADLIB environment variable (UNIX)" on page 3-72

Table 3-1. Uses for environment variables (continued)

Functional category	Environment variable
Configuration	Specify the name of the active that holds configuration parameters: "ONCONFIG environment variable" on page 3-60 Ignore specified environment variable settings: "ENVIGNORE environment variable (UNIX)" on page 3-38 Specify the default database server: "INFORMIXSERVER environment variable" on page 3-54 Specify the dbspaces in which temporary tables are built: "DBSPACETEMP environment variable" on page 3-30 Manage query optimizer directives: "IFX_DIRECTIVES environment variable" on page 3-40 and "IFX_EXTDIRECTIVES environment variable" on page 3-41 Modify the value of the OPTCOMPIND configuration parameter: "OPTCOMPIND environment variable" on page 3-61 Specify the query performance goal for the optimizer: "OPT_GOAL environment variable (UNIX)" on page 3-63 Specify the degree of parallelism that the database server uses: "PDQPRIORITY environment variable" on page 3-64 Specify the stack size that is applied to all client processes: "INFORMIXSTACKSIZE environment variable" on page 3-56
Connecting	Set the maximum number of *additional* connection attempts: "INFORMIXCONRETRY environment variable" on page 3-51 Set connect time information: "INFORMIXCONTIME environment variable" on page 3-52 Specify the default database server to for connections: "INFORMIXSERVER environment variable" on page 3-54 Specify the location of connection information: "INFORMIXSQLHOSTS environment variable" on page 3-55
Connection Manager	Specify the location of the Connection Manager configuration file:"CMCONFIG environment variable" on page 3-17
Data distributions	Manage the amount of system disk space that the UPDATE STATISTICS statement can use: "DBUPSPACE environment variable" on page 3-35
Database locale	Manage locale information: GLS-related environment variables

Table 3-1. Uses for environment variables (continued)

Functional category	**Environment variable**
Database server	Specify servers for connections: “INFORMIXSERVER environment variable” on page 3-54 Set the locale for file I/O: GLS-related environment variables Specify the name of the active file that holds configuration parameters: “ONCONFIG environment variable” on page 3-60 Manage parallel sorting: “PSORT_DBTEMP environment variable” on page 3-68 and “PSORT_NPROCS environment variable” on page 3-68 Manage parallelism: “PDQPRIORITY environment variable” on page 3-64 Manage role separation: “INF_ROLE_SEP environment variable” on page 3-57 Manage shared memory: “INFORMIXSHMBASE environment variable (UNIX)” on page 3-55 Manage stack size: “INFORMIXSTACKSIZE environment variable” on page 3-56 Manage temporary tables: “DBSPACETEMP environment variable” on page 3-30, “DBTEMP environment variable” on page 3-32, and “PSORT_DBTEMP environment variable” on page 3-68 Manage variable-length packets: “IFX_PAD_VARCHAR environment variable” on page 3-46
Date and time values, formats	Manage date and time information: “DBCENTURY environment variable” on page 3-20, “DBDATE environment variable” on page 3-22, “DBTIME environment variable” on page 3-32, GLS-related environment variables (**GL_DATE** and **GL_DATETIME**), The USE_DTENV environment variable, and “TZ environment variable” on page 3-73
DB-Access utility	Manage the database server and DB-Access: “DBANSIWARN environment variable” on page 3-19, “DBDELIMITER environment variable” on page 3-24, “DBEDIT environment variable” on page 3-25. “DBFLTMASK environment variable” on page 3-25, “DBPATH environment variable” on page 3-28, “FET_BUF_SIZE environment variable” on page 3-39, “INFORMIXSERVER environment variable” on page 3-54, “INFORMIXTERM environment variable (UNIX)” on page 3-56, “TERM environment variable (UNIX)” on page 3-71, “TERMCAP environment variable (UNIX)” on page 3-72, and “TERMINFO environment variable (UNIX)” on page 3-72
dbexport utility	Set the field delimiter: “DBDELIMITER environment variable” on page 3-24
Delimited identifiers	Set the field delimiter used with the **dbexport** utility and with the LOAD and UNLOAD statements: “DBDELIMITER environment variable” on page 3-24
Disk space	Manage the amount of system disk space and memory that the UPDATE STATISTICS MEDIUM or HIGH statement can use: “DBUPSPACE environment variable” on page 3-35
Editor	Specify the text editor to use with SQL statements and command files in DB-Access: “DBEDIT environment variable” on page 3-25
Enterprise Replication	Specify information for Enterprise Replication: Enterprise Replication configuration parameter and environment variable reference

Table 3-1. Uses for environment variables (continued)

Functional category	Environment variable
ESQL/C	Specify ANSI compliance: "DBANSIWARN environment variable" on page 3-19 Specify the filename or pathname of the C compiler to use with ESQL/C: "INFORMIXC environment variable (UNIX)" on page 3-50 Set delimited identifiers: "DELIMIDENT environment variable" on page 3-37 Specify multibyte characters and locale information GLS-related environment variables (**CLIENT_LOCALE**, **ESQLMF**, and **GL_USER**) Specify information for multithreaded applications: "THREADLIB environment variable (UNIX)" on page 3-72 Specify the default compilation order: "CPFIRST environment variable" on page 3-17
Executable programs	Specify the directories to search for executable programs: "PATH environment variable" on page 3-63
Fetch buffer size	Set buffer size information: "FET_BUF_SIZE environment variable" on page 3-39
Filenames: multibyte	GLS-related environment variables (**GLS8BITFSYS**)
Files: field delimiter	Set the field delimiter: "DBDELIMITER environment variable" on page 3-24
Files: installation	Specify the directory that contains the subdirectories in which your product files are installed: "INFORMIXDIR environment variable" on page 3-54
Files: locale	Specify locale information: GLS-related environment variables (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**)
Files: map for C++	Specify the pathname of the map file for C++ programs: "INFORMIXCPPMAP environment variable" on page 3-54
Files: message	Specify the subdirectory of **$INFORMIXDIR** or the pathname of the directory that contains the compiled message files that the database server uses: "DBLANG environment variable" on page 3-26
Files: temporary	"DBSPACETEMP environment variable" on page 3-30
Files: temporary	Specify a directory for temporary files: "DBTEMP environment variable" on page 3-32
Files: temporary sorting	Specify the location of temporary files used for sorting: "PSORT_DBTEMP environment variable" on page 3-68
Files: `termcap`, `terminfo`	Specify terminal information: "INFORMIXTERM environment variable (UNIX)" on page 3-56, "TERM environment variable (UNIX)" on page 3-71, "TERMCAP environment variable (UNIX)" on page 3-72, and "TERMINFO environment variable (UNIX)" on page 3-72
Format: date and time	Define the format for date and time information: "DBCENTURY environment variable" on page 3-20, "DBDATE environment variable" on page 3-22, "DBTIME environment variable" on page 3-32, GLS-related environment variables (**GL_DATE** and **GL_DATETIME**), The USE_DTENV environment variable, and "TZ environment variable" on page 3-73
Format: private-use characters	Set the display width for characters in Unicode Private-Use Area (PUA) ranges: GLS-related environment variables (IFX_PUA_DISPLAY_MAPPING)
Format: money	Define the format for money information: "DBMONEY environment variable" on page 3-27and GLS-related environment variables
Gateways	Set information for Gateways: "DBTEMP environment variable" on page 3-32
High-Performance Loader	Specify information for the High-Performance Loader: "DBONPLOAD environment variable" on page 3-27, , "PLOAD_LO_PATH environment variable" on page 3-65. and "PLOAD_SHMBASE environment variable" on page 3-65

Table 3-1. Uses for environment variables (continued)

Functional category	Environment variable
Identifiers	Specify field delimiters: "DELIMIDENT environment variable" on page 3-37 Specify information for identifiers longer than 18 bytes: "IFX_LONGID environment variable" on page 3-43 Specify information for multibyte characters: GLS-related environment variables (**CLIENT_LOCALE** and **ESQLMF**)
Installation	Specify the directory that contains the subdirectories in which your product files are installed: "INFORMIXDIR environment variable" on page 3-54 Specify which directories to search for executable programs: "PATH environment variable" on page 3-63
JDBC	Manage environment variables used with JDBC: Informix environment variables with the IBM Informix JDBC Driver
Language environment	Specify language and locale information: "DBLANG environment variable" on page 3-26 and GLS-related environment variables
Libraries	Specify paths for libraries: "LD_LIBRARY_PATH environment variable (UNIX)" on page 3-59, "LIBPATH environment variable (UNIX)" on page 3-59, and "SHLIB_PATH environment variable (UNIX)" on page 3-70
Locale	Define client, server, and database locale information: GLS-related environment variables (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**)
Lock mode	Set the default lock mode for database tables that are created without specifying the LOCKMODE PAGE or LOCKMODE ROW keywords: "IFX_DEF_TABLE_LOCKMODE environment variable" on page 3-40
Long Identifiers	Specify information for identifiers longer than 18 bytes: "IFX_LONGID environment variable" on page 3-43
Map file for C++	Specify the pathname of the map file for C++ programs: "INFORMIXCPPMAP environment variable" on page 3-54
Message chaining	Enable or disable optimized message transfers (message chaining) for IBM Informix ESQL/C: "OPTMSG environment variable" on page 3-62
Message files	Specify the directory that contains compiled message files: "DBLANG environment variable" on page 3-26
Money format	Define the format for money information: "DBMONEY environment variable" on page 3-27and GLS-related environment variables
Multibyte characters	Specify information for multibyte characters: GLS-related environment variables (**CLIENT_LOCALE**, **DB_LOCALE**, **SERVER_LOCALE**, and **GL_USEGLU**)
Multibyte filter	Specify Informix ESQL/C multibyte filter information: GLS-related environment variables (**ESQLMF**)
Multithreaded applications	Specify information for compiling multithreaded IBM Informix ESQL/C applications: "THREADLIB environment variable (UNIX)" on page 3-72
Network	Specify network information: "DBPATH environment variable" on page 3-28
Nondefault locale	Define client, server, and database locale information: GLS-related environment variables (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**)
ON-Bar utility	Optimize the deduplication capabilities for storage managers: IFX_BAR_USE_DEDUP environment variable Disable the ability to replicate, import, or export backup objects among TSM servers: IFX_TSM_OBJINFO_OFF environment variable
ONCONFIG parameters	Specify the name of the file that holds configuration parameters: "ONCONFIG environment variable" on page 3-60

Table 3-1. Uses for environment variables (continued)

Functional category	Environment variable
oninit output (Windows only)	Specify a path and file for **oninit** output: "ONINIT_STDOUT environment variable (Windows)" on page 3-60
Optimization: directives	Manage query optimizer directives: "IFX_DIRECTIVES environment variable" on page 3-40 and "IFX_EXTDIRECTIVES environment variable" on page 3-41
Optimization: message transfers	Enable or disable optimized message transfers (message chaining) for IBM Informix ESQL/C: "OPTMSG environment variable" on page 3-62
Optimization: join method	Modify the value of the OPTCOMPIND configuration parameter: "OPTCOMPIND environment variable" on page 3-61
Optimization: performance goal	Specify the query performance goal for the optimizer: "OPT_GOAL environment variable (UNIX)" on page 3-63
OPTOFC feature	Enable optimize-OPEN-FETCH-CLOSE functionality: "OPTOFC environment variable" on page 3-62
PAM authentication for MongoDB clients	Enable PAM authentication for MongoDB clients: "IFMXMONGOAUTH environment variable" on page 3-39
Path name: **archecker** configuration file	Specify the full path name for the **archecker** configuration file: AC_CONFIG file environment variable
Path name: C compiler	Specify the filename or pathname of the C compiler: "INFORMIXC environment variable (UNIX)" on page 3-50
Path name: database files	Specify database server file and path information: "DBPATH environment variable" on page 3-28
Path name: executable programs	Specify directories to search for executable programs: "PATH environment variable" on page 3-63
Path name: HPL smart-large-object handles	Specify the pathname for smart-large-object handles: "PLOAD_LO_PATH environment variable" on page 3-65
Path name: installation	Specify the directory that contains the subdirectories in which your product files are installed: "INFORMIXDIR environment variable" on page 3-54
Path name: libraries	Specify paths for libraries: "LD_LIBRARY_PATH environment variable (UNIX)" on page 3-59, "LIBPATH environment variable (UNIX)" on page 3-59, and "SHLIB_PATH environment variable (UNIX)" on page 3-70
Path name: message files	Specify the directory that contains compiled message files: "DBLANG environment variable" on page 3-26 and GLS-related environment variables
Path name: parallel sorting	Specify the location of temporary files for sorts: "PSORT_DBTEMP environment variable" on page 3-68
IBM Informix Primary Storage Manager	Manage the storage manager: "PSM_ACT_LOG environment variable" on page 3-66. "PSM_CATALOG_PATH environment variable" on page 3-66, "PSM_DBS_POOL environment variable" on page 3-66, "PSM_DEBUG environment variable" on page 3-67, "PSM_DEBUG_LOG environment variable" on page 3-67, and "PSM_LOG_POOL environment variable" on page 3-67
Preserve owner name case	Set the case of owner names: "ANSIOWNER environment variable" on page 3-16
Printing	Specify the default printing program: "DBPRINT environment variable" on page 3-30
Privileges	Configure role separation: "INF_ROLE_SEP environment variable" on page 3-57

Table 3-1. Uses for environment variables (continued)

Functional category	**Environment variable**
Query: optimization	Manage query optimizer directives: "IFX_DIRECTIVES environment variable" on page 3-40 and "IFX_EXTDIRECTIVES environment variable" on page 3-41 Modify the value of the OPTCOMPIND configuration parameter: "OPTCOMPIND environment variable" on page 3-61 Specify the query performance goal for the optimizer: "OPT_GOAL environment variable (UNIX)" on page 3-63 Specify user-defined data types can use to estimate the cost of an R-tree index for queries on UDT columns"RTREE_COST_ADJUST_VALUE environment variable" on page 3-69
Query: prioritization	Specify the degree of parallelism that the database server uses: "PDQPRIORITY environment variable" on page 3-64
Remote shell	Specify information that overrides the default remote shell for performing remote tape operations: "DBREMOTECMD environment variable (UNIX)" on page 3-30
Role separation	Configure role separation: "INF_ROLE_SEP environment variable" on page 3-57
Rolled-back transactions	Manage what the DB-Access utility does when an error occurs: "DBACCNOIGN environment variable" on page 3-18 Specify whether an internal rollback of a global transaction frees the transaction: "IFX_XASTDCOMPLIANCE_XAEND environment variable" on page 3-48
Server locale	Define the locale of your database server: GLS-related environment variables **SERVER_LOCALE**
Shared memory	Specify where shared-memory segments are attached to the client process: "INFORMIXSHMBASE environment variable (UNIX)" on page 3-55 Specify the shared-memory address for High Performance Loader (HPL) processes:"PLOAD_SHMBASE environment variable" on page 3-65
Shell: remote	Specify information that overrides the default remote shell for performing remote tape operations: "DBREMOTECMD environment variable (UNIX)" on page 3-30
Shell: search path	Specify which directories to search for executable programs: "PATH environment variable" on page 3-63
Sorting	Specify the location of temporary files for sorts: "PSORT_DBTEMP environment variable" on page 3-68 Allocate more threads for sorting: "PSORT_NPROCS environment variable" on page 3-68

Table 3-1. Uses for environment variables (continued)

Functional category	Environment variable
SQL statements	Specify information for caching: "STMT_CACHE environment variable" on page 3-71 Specify connection information: "INFORMIXCONRETRY environment variable" on page 3-51, "INFORMIXCONTIME environment variable" on page 3-52, and "INFORMIXSERVER environment variable" on page 3-54 Specify information for CREATE TEMP TABLE operations: "DBSPACETEMP environment variable" on page 3-30 Specify information for DESCRIBE FOR UPDATE operations: "IFX_UPDDESC environment variable" on page 3-47 Specify information for LOAD and UNLOAD operations: "DBDELIMITER environment variable" on page 3-24 and "DBBLOBBUF environment variable" on page 3-20 Specify information for SELECT INTO TEMP operations: "DBSPACETEMP environment variable" on page 3-30 Specify information for SET PDQPRIORITY operations: "PDQPRIORITY environment variable" on page 3-64 Specify information for SET STMT_CACHE operations Specify information for UPDATE STATISTICS operations: "DBUPSPACE environment variable" on page 3-35
Stack size	Define the stack size that is applied to client processes: "INFORMIXSTACKSIZE environment variable" on page 3-56
Temporary tables	Define information for temporary tables: "DBSPACETEMP environment variable" on page 3-30, "DBTEMP environment variable" on page 3-32, and "PSORT_DBTEMP environment variable" on page 3-68
Terminal handling	Specify terminal information: "INFORMIXTERM environment variable (UNIX)" on page 3-56, "TERM environment variable (UNIX)" on page 3-71, "TERMCAP environment variable (UNIX)" on page 3-72, and "TERMINFO environment variable (UNIX)" on page 3-72
Time-limited software license	Set information for trial or evaluation software warning messages: "IFX_NO_TIMELIMIT_WARNING environment variable" on page 3-45
Variables: overriding	Deactivate some specified environment variable settings: "ENVIGNORE environment variable (UNIX)" on page 3-38
Virtual memory segments on large pages	Specify whether the database server can use large pages on platforms where the hardware and the operating system support large pages of shared memory: "IFX_LARGE_PAGES environment variable" on page 3-42
Year values (abbreviated)	Specify how to expand DATE and DATETIME values that are entered as abbreviated year values: "DBCENTURY environment variable" on page 3-20

ANSIOWNER environment variable

In an ANSI-compliant database, you can prevent the default behavior of upshifting lowercase letters in owner names that are not delimited by quotation marks by setting the **ANSIOWNER** environment variable to 1.

```
►►──setenv──ANSIOWNER── 1──────────────────────────────────────────►◄
```

To prevent upshifting of lowercase letters in owner names in an ANSI-compliant database, you must set **ANSIOWNER** before you initialize IBM Informix.

The following table shows how an ANSI-compliant database of IBM Informix stores or reads the specified name of a database object called **oblong** if you were the owner of **oblong** and your **userid** (in all lowercase letters) were **owen**:

Table 3-2. Lettercase of implicit, unquoted, and quoted owner names, with and without ANSIOWNER

Owner Format	Specification	ANSIOWNER = 1	ANSIOWNER Not Set
Implicit:	oblong	owen.oblong	OWEN.oblong
Unquoted:	owen.oblong	owen.oblong	OWEN.oblong
Quoted:	'owen'.oblong	owen.oblong	owen.oblong

Because they do not match the lettercase of your **userid**, any SQL statements that specified the formats that are stored as **OWEN.oblong** would fail with errors.

CPFIRST environment variable

Use the **CPFIRST** environment variable to specify the default compilation order for all IBM Informix ESQL/C source files in your programming environment.

```
►►──setenv──CPFIRST──┬─TRUE──┬──────────────────────────────────────────►◄
                     └─FALSE─┘
```

When you compile Informix ESQL/C programs with **CPFIRST** not set, the Informix ESQL/C preprocessor runs first, by default, on the program source file and then passes the resulting file to the C language preprocessor and compiler. You can, however, compile the Informix ESQL/C program source file in the following order:

1. Run the C preprocessor
2. Run the Informix ESQL/C preprocessor
3. Run the C compiler and linker

To use a nondefault compilation order for a specific program, you can either give the program source file a `.ecp` extension, run the `-cp` option with the **esql** command on a program source file with a `.ec` extension, or set **CPFIRST**.

Set **CPFIRST** to `TRUE` (uppercase only) to run the C preprocessor before the Informix ESQL/C preprocessor on all Informix ESQL/C source files in your environment, irrespective of whether the `-cp` option is passed to the **esql** command or the source files have the `.ec` or the `.ecp` extension.

To restore the default order on a system where the **CPFIRST** environment variable has been set to `TRUE`, you can set **CPFIRST** to `FALSE`. On UNIX systems that support the C shell, the following command has the same effect:

```
unsetenv CPFIRST
```

CMCONFIG environment variable

Set the **CMCONFIG** environment variable to specify the location of the Connection Manager configuration file. You use the configuration file to specify service level agreements and other Connection Manager configuration options.

```
►►—setenv—CMCONFIG—path/file_name——————————————————————►◄
```

path/file_name
: is the full path and file name of a Connection Manager configuration file.

If the CMCONFIG environment variable is not set and the configuration file name is not specified on the **oncmsm** utility command line, the Connection Manager attempts to load the file from the following path and file name:

```
$INFORMIXDIR/etc/cmsm.cfg
```

Examples

Suppose the CMCONFIG environment variable points to a valid path and file name of a Connection Manager configuration file. To reload a Connection Manager instance using the configuration file specified in the shell environment enter the following command:

```
./oncmsm -r
```

To shut down a Connection Manager instance using the configuration file specified in the shell environment:

```
./oncmsm -k
```

Related information:

The oncmsm utility

Example of configuring connection management for a high-availability cluster

DBACCNOIGN environment variable

Use the **DBACCNOIGN** environment variable to specify the behavior of the DB-Access utility when specified errors occurs.

The **DBACCNOIGN** environment variable affects the behavior of the DB-Access utility if an error occurs under one of the following circumstances:

- You run DB-Access in non-menu mode.
- In IBM Informix only, you execute the LOAD command with DB-Access in menu mode.

Set the **DBACCNOIGN** environment variable to 1 to roll back an incomplete transaction if an error occurs while you run the DB-Access utility under either of the preceding conditions.

```
►►—setenv—DBACCNOIGN—1—————————————————————————————————►◄
```

For example, assume DB-Access runs the following SQL commands:

```
DATABASE mystore
BEGIN WORK

INSERT INTO receipts VALUES (cust1, 10)
INSERT INTO receipt VALUES (cust1, 20)
INSERT INTO receipts VALUES (cust1, 30)

UPDATE customer
   SET balance =
      (SELECT (balance-60)
      FROM customer WHERE custid = 'cust1')
   WHERE custid = 'cust1
COMMIT WORK
```

Here, one statement has a misspelled table name: the **receipt** table does not exist. If **DBACCNOIGN** is not set in your environment, DB-Access inserts two records into the **receipts** table and updates the **customer** table. Now, the decrease in the **customer** balance exceeds the sum of the inserted receipts.

But if **DBACCNOIGN** is set to 1, messages open that indicate that DB-Access rolled back all the INSERT and UPDATE statements. The messages also identify the cause of the error so that you can resolve the problem.

LOAD statement example when DBACCNOIGN is set

You can set the **DBACCNOIGN** environment variable to protect data integrity during a LOAD statement, even if DB-Access runs the LOAD statement in menu mode.

Assume you execute the LOAD statement from the DB-Access SQL menu. Forty-nine rows of data load correctly, but the 50th row contains an invalid value that causes an error. If you set **DBACCNOIGN** to 1, the database server does not insert the forty-nine previous rows into the database. If **DBACCNOIGN** is not set, the database server inserts the first 49 rows.

DBANSIWARN environment variable

Use the **DBANSIWARN** environment variable to indicate that you want to check for IBM Informix extensions to ANSI-standard SQL syntax.

Unlike most environment variables, you are not required to set

```
DBANSIWARN
```

to a value. You can set it to any value or to no value.

```
►►──setenv──DBANSIWARN─────────────────────────────────────────────►◄
```

Running DB-Access with **DBANSIWARN** set is functionally equivalent to including the **-ansi** flag when you invoke DB-Access (or any IBM Informix product that recognizes the **-ansi** flag) from the command line. If you set **DBANSIWARN** before you run DB-Access, any syntax-extension warnings are displayed on the screen within the SQL menu.

At runtime, the **DBANSIWARN** environment variable causes the sixth character of the **sqlwarn** array in the SQL Communication Area (SQLCA) to be set to `W` when a statement is executed that is recognized as including any IBM Informix extension to the ANSI/ISO standard for SQL syntax.

For details on SQLCA, see the *IBM Informix ESQL/C Programmer's Manual*.

After you set **DBANSIWARN**, IBM Informix extension checking is automatic until you log out or unset **DBANSIWARN**. To turn off IBM Informix extension checking, you can disable **DBANSIWARN** with this command:

```
unsetenv DBANSIWARN
```

DBBLOBBUF environment variable

Use the **DBBLOBBUF** environment variable to control whether TEXT or BYTE values are stored temporarily in memory or in a file while being processed by the UNLOAD statement. **DBBLOBBUF** affects only the UNLOAD statement.

```
►►──setenv──DBBLOBBUF──size──────────────────────────────────────────►◄
```

size represents the maximum size of TEXT or BYTE data in KB.

If the TEXT or BYTE data size is smaller than the default of 10 KB (or the setting of **DBBLOBBUF**), the TEXT or BYTE value is temporarily stored in memory. If the data size is larger than the default or the **DBBLOBBUF** setting, the data value is written to a temporary file. For instance, to set a buffer size of 15 KB, set **DBBLOBBUF** as in the following example:

```
setenv DBBLOBBUF 15
```

Here any TEXT or BYTE value smaller than 15 KB is stored temporarily in memory. Values larger than 15 KB are stored temporarily in a file.

DBCENTURY environment variable

Use the **DBCENTURY** environment variable to specify how to expand literal DATE and DATETIME values that are entered with abbreviated year values. To avoid problems in expanding abbreviated years, applications should require entry of 4-digit years, and should always display years as four digits.

```
                         ┌─R──────┐
►►──setenv──DBCENTURY────┼─F──────┼──────────────────────────────────►◄
                         └──┬─C─┬─┘
                            └─P─┘
```

When **DBCENTURY** is not set (or is set to `R`), the first two digits of the current year are used to expand 2-digit year values. For example, if today's date is 09/30/2003, then the abbreviated date 12/31/99 expands to 12/31/2099, and the abbreviated date 12/31/00 expands to 12/31/2000.

The R, P, F, and C settings determine algorithms for expanding two-digit years.

Setting	Algorithm
R = Current	Use the first two digits of the current year to expand the year value.
P = Past	Expanded dates are created by prefixing the abbreviated year value with 19 and 20. Both dates are compared to the current date, and the most recent date that is earlier than the current date is used.
F = Future	Expanded dates are created by prefixing the abbreviated year value with 20 and 21. Both dates are compared to the current date, and the earliest date that is later than the current date is used.
C = Closest	Expanded dates are created by prefixing the abbreviated year value with 19, 20, and 21. These three dates are compared to the current date, and the date that is closest to the current date is used.

Settings are case sensitive, and no error is issued for invalid settings. If you enter `f` (for example), then the default (`R`) setting takes effect. The `P` and `F` settings cannot return the current date, which is not in the past or future.

Years entered as a single digit are prefixed with 0 and then expanded. Three-digit years are not expanded. Pad years earlier than 100 with leading zeros.

Related reference:

"DATETIME data type" on page 2-12

Examples of expanding year values

The examples in this topic illustrate how various settings of **DBCENTURY** cause abbreviated years to be expanded in DATE and DATETIME values.

DBCENTURY = P

```
Example data type: DATE
Current date: 4/6/2003
User enters: 1/1/1
Prefix with "19" expansion : 1/1/1901
Prefix with "20" expansion: 1/1/2001
Analysis: Both are prior to current date, but 1/1/2001 is closer to
 current date.
```

Important: The effect of **DBCENTURY** depends on the current date from the system clock-calendar. Thus, 1/1/1, the abbreviated date in this example, would instead be expanded to 1/1/1901 if the current date were 1/1/2001 and **DBCENTURY** = P.

DBCENTURY = F

```
Example data type: DATETIME year to month
Current date: 5/7/2005
User enters: 1-1
Prefix with "20" expansion: 2001-1
Prefix with "21" expansion: 2101-1
Analysis: Only date 2101-1 is after the current date, so it is chosen.
```

DBCENTURY = C

```
Example data type: DATE
Current date: 4/6/2000
User enters: 1/1/1
Prefix with "19" expansion : 1/1/1901
Prefix with "20" expansion: 1/1/2001
Prefix with "21" expansion: 1/1/2101
Analysis: Here 1/1/2001 is closest to the current date, so it is chosen.
```

DBCENTURY = R or DBCENTURY Not Set

```
Example data type: DATETIME year to month
Current date: 4/6/2000
User enters: 1-1
Prefix with "20" expansion: 2001-1

Example data type: DATE
Current date: 4/6/2003
User enters: 0/1/1
Prefix with "20" expansion: 2000/1
Analysis: In both examples, the Prefix with "20" algorithm is used.
```

Setting **DBCENTURY** does not affect IBM Informix products when the locale specifies a non-Gregorian calendar, such as Hebrew or Islamic calendars. The leading digits of the current year are used for alternative calendar systems when the year is abbreviated.

Abbreviated years and expressions in database objects

When an expression in a database object (including a check constraint, fragmentation expression, SPL routine, trigger, or UDR) contains a literal date or DATETIME value in which the year has one or two digits, the database server evaluates the expression using the setting that **DBCENTURY** (and other relevant environment variables) had when the database object was created (or was last modified).

If **DBCENTURY** has been reset to a new value, the new value is ignored when the abbreviated year is expanded.

For example, suppose a user creates a table and defines the following check constraint on a column named **birthdate**:

```
birthdate < '09/25/50'
```

The expression is interpreted according to the value of **DBCENTURY** when the constraint was defined. If the table that contains the **birthdate** column is created on 09/23/2000 and **DBCENTURY** =`C`, the check constraint expression is consistently interpreted as `birthdate < '09/25/1950'` when inserts or updates are performed on the **birthdate** column. Even if different values of **DBCENTURY** are set when users perform inserts or updates on the **birthdate** column, the constraint expression is interpreted according to the setting at the time when the check constraint was defined (or was last modified).

Database objects created on some earlier versions of IBM Informix do not support the priority of creation-time settings.

For legacy objects to acquire this feature

1. Drop the objects.
2. Recreate them (or for fragmentation expressions, detach them and then reattach them).

After the objects are redefined, date literals within expressions of the objects will be interpreted according to the environment at the time when the object was created or was last modified. Otherwise, their behavior will depend on the runtime environment and might become inconsistent if this changes.

Administration of a database that includes a mix of legacy objects and new objects might become difficult because of differences between the new and the old behavior for evaluating date expressions. To avoid this, it is recommended that you redefine any legacy objects.

The value of **DBCENTURY** and the current date are not the only factors that determine how the database server interprets date and DATETIME values. The **DBDATE**, **DBTIME**, **GL_DATE**, and **GL_DATETIME** environment variables can also influence how dates are interpreted. For information about **GL_DATE** and **GL_DATETIME**, see the *IBM Informix GLS User's Guide*.

Important: The behavior of **DBCENTURY** for IBM Informix is not compatible with earlier versions.

DBDATE environment variable

Use the **DBDATE** environment variable to specify the end-user formats of DATE values.

On UNIX systems that use the C shell, set **DBDATE** with this syntax.

```
                     ┌─MD─┐   ┌─Y4─┐   ┌─/─┐
►►──setenv──DBDATE───┼─DM─┼───┼─Y2─┼───┼─-─┼──────────────────────────────────►◄
                     ├─Y4─┤   ├─MD─┤   ├─.─┤
                     └─Y2─┘   └─DM─┘   ├─.─┤
                                       └─0─┘
```

The following formatting symbols are valid in the **DBDATE** setting:

- . /	are characters that can exist as separators in a date format.
0	indicates that no separator is displayed between time units.
D, M	are characters that represent the day and the month.
Y2, Y4	are characters that represent the year and the precision of the year.

Some East Asian locales support additional syntax for era-based dates.

DBDATE can specify the following attributes of the display format:
- The order of time units (the month, day, and year) in a date
- Whether the year is shown as two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year time units

For the U.S. English locale, the default for **DBDATE** is `MDY4/`, where `M` represents the month, `D` represents the day, `Y4` represents a four-digit year, and slash (**/**) is the time-units separator (for example, `01/08/2011`). Other valid characters for the separator are a hyphen (**-**), a period (**.**), or a zero (0). To indicate no separator, use the zero. The slash (**/**) is used by default if you attempt to specify a character other than a hyphen, period, or zero as a separator, or if you do not include any separator in the **DBDATE** specification.

If **DBDATE** is not set on the client, any **DBDATE** setting on the database server overrides the `MDY4/` default on the client. If **DBDATE** is set on the client, that value (rather than the setting on the database server) is used by the client.

The following table shows some examples of valid **DBDATE** settings and their corresponding displays for the date 8 January, 2011:

DBDATE Setting	**Representation of January 8, 2011:**		**DBDATE Setting**	**Representation of January 8, 2011:**
MDY4/	01/08/2011		Y2DM.	11.08.01
DMY2-	08-01-11		MDY20	010811
MDY4	01/08/2011		Y4MD*	2011/01/08

Formats `Y4MD*` (because asterisk is not a valid separator) and `MDY4` (with no separator defined) both display the default symbol (slash) as the separator.

Important: If you use the Y2 format, the setting of the **DBCENTURY** environment variable can also affect how literal DATE values are evaluated in data entry.

Also, certain routines that *IBM Informix ESQL/C* calls can use the **DBTIME** variable, rather than **DBDATE**, to set DATETIME formats to international specifications. For more information, see the discussion of the **DBTIME** environment variable in "DBTIME environment variable" on page 3-32 and in the *IBM Informix ESQL/C Programmer's Manual*.

The setting of the **DBDATE** variable takes precedence over that of the **GL_DATE** environment variable, and over any default DATE format that **CLIENT_LOCALE** specifies. For information about **GL_DATE** and **CLIENT_LOCALE**, see the *IBM Informix GLS User's Guide*.

End-user formats affect the following contexts:
- When you display DATE values, IBM Informix products use the **DBDATE** environment variable to format the output.
- During data entry of DATE values, IBM Informix products use the **DBDATE** environment variable to interpret the input.

For example, if you specify a literal DATE value in an INSERT statement, the database server expects this literal value to be compatible with the format that **DBDATE** specifies. Similarly, the database server interprets the date that you specify as the argument to the **DATE**( ) function to be in **DBDATE** format.

DATE expressions in database objects

When an expression in a database object (including a check constraint, fragmentation expression, SPL routine, trigger, or UDR) contains a literal date value, the database server evaluates the expression using the setting that **DBDATE** (or other relevant environment variables) had when the database object was created (or was last modified). If **DBDATE** has been reset to a new value, the new value is ignored when the literal DATE is evaluated.

For example, suppose **DBDATE** is set to `MDY2/` and a user creates a table with the following check constraint on the column **orderdate**:

```
orderdate < '06/25/98'
```

The date of the preceding expression is formatted according to the value of **DBDATE** when the constraint is defined. The check constraint expression is interpreted as `orderdate < '06/25/98'` regardless of the value of **DBDATE** during inserts or updates on the **orderdate** column. Suppose **DBDATE** is reset to `DMY2/` when a user inserts the value `'30/01/98'` into the **orderdate** column. The date value inserted uses the date format `DMY2/`, whereas the check constraint expression uses the date format `MDY2/`.

See "Abbreviated years and expressions in database objects" on page 3-22 for a discussion of legacy objects from earlier versions of IBM Informix that are always evaluated according to the runtime environment. That section describes how to redefine objects so that dates are interpreted according to environment variable settings that were in effect when the object was defined (or when the object was last modified).

Important: The behavior of **DBDATE** for IBM Informix is not compatible with earlier versions.

DBDELIMITER environment variable

Set the **DBDELIMITER** environment variable to specify the field delimiter used with the **dbexport** utility and with the LOAD and UNLOAD statements.

```
►►──setenv──DBDELIMITER──'delimiter'──────────────────────────────►◄
```

delimiter
is the field delimiter for unloaded data files.

The *delimiter* can be any single character, except those in the following list:
- Hexadecimal digits (0 through 9,a through f, A through F)
- Newline or `CTRL-J`
- The backslash (\ ) symbol

The vertical bar (| = ASCII 124) is the default. To change the field delimiter to a plus (+) symbol, for example, you can set DBDELIMITER as follows:

```
setenv DBDELIMITER '+'
```

DBEDIT environment variable

Use the **DBEDIT** environment variable to specify the text editor to use with SQL statements and command files in DB-Access.

If **DBEDIT** is set, the specified text editor is invoked automatically. If **DBEDIT** is not, set you are prompted to specify a text editor as the default for the rest of the session.

```
►►──setenv──DBEDIT──editor────────────────────────────────────────►◄
```

editor is the name of the text editor you want to use.

For most UNIX systems, the default text editor is **vi**. If you use another text editor, be sure that it creates flat ASCII files. Some word processors in *document mode* introduce printer control characters that can interfere with the operation of your IBM Informix product.

To specify the EMACS text editor, set **DBEDIT** with the following command:

```
setenv DBEDIT emacs
```

DBFLTMASK environment variable

The DB-Access utility displays the floating-point values of data types FLOAT, SMALLFLOAT, and DECIMAL(*p*) within a 14-character buffer. By default, DB-Access displays as many digits to the right of the decimal point as will fit into this character buffer. Therefore, the actual number of decimal digits that DB-Access displays depends on the size of the floating-point value.

To reduce the number of digits displayed to the right of the decimal point in floating-point values, set **DBFLTMASK** to the specified number of digits.

```
►►──setenv──DBFLTMASK──scale──────────────────────────────────────►◄
```

scale is the number of decimal digits that you want the IBM Informix client application to display in the floating-point values. Here *scale* must be smaller than 16, the default number of digits displayed.

If the floating-point value contains more digits to the right of the decimal than **DBFLTMASK** specifies, DB-Access rounds the value to the specified number of digits. If the floating-point value contains fewer digits to the right of the decimal, DB-Access pads the value with zeros. If you set **DBFLTMASK** to a value greater than can fit into the 14-character buffer, however, DB-Access rounds the value to the number of digits that can fit.

DBLANG environment variable

Use the **DBLANG** environment variable to specify the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the compiled message files that IBM Informix products use.

```
►►──setenv──DBLANG──┬─relative_path─┬──────────────────────────────────►◄
                    └─full_path─────┘
```

relative_path
is a subdirectory of **$INFORMIXDIR**.

full_path
is the pathname to the compiled message files.

By default, IBM Informix products put compiled messages in a locale-specific subdirectory of the **$INFORMIXDIR/msg** directory. These compiled message files have the file extension **.iem**. If you want to use a message directory other than **$INFORMIXDIR/msg**, where, for example, you can store message files that you create, you must perform the following steps:

To use a message directory other than $INFORMIXDIR/msg

1. Use the **mkdir** command to create the appropriate directory for the message files.

   You can make this directory under the directory **$INFORMIXDIR** or **$INFORMIXDIR/msg**, or you can make it under any other directory.
2. Set the owner and group of the new directory to **informix** and the access permission for this directory to 755.
3. Set the **DBLANG** environment variable to the new directory. If this is a subdirectory of **$INFORMIXDIR** or **$INFORMIXDIR/msg**, then you need only list the relative path to the new directory. Otherwise, you must specify the full pathname of the directory.
4. Copy the **.iem** files or the message files that you created to the new message directory that **$DBLANG** specifies.

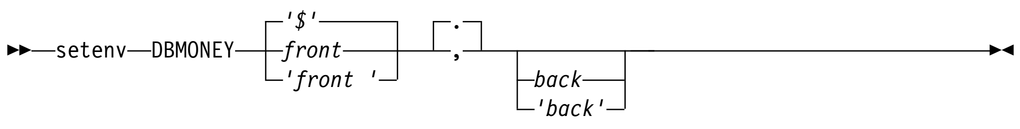
   All the files in the message directory should have the owner and group **informix** and access permission 644.

IBM Informix products that use the default U.S. English locale search for message files in the following order:
1. In **$DBLANG**, if **DBLANG** is set to a full pathname
2. In **$INFORMIXDIR/msg/$DBLANG**, if **DBLANG** is set to a relative pathname
3. In **$INFORMIXDIR/$DBLANG**, if **DBLANG** is set to a relative pathname
4. In **$INFORMIXDIR/msg/en_us/0333**
5. In **$INFORMIXDIR/msg/en_us.8859-1**
6. In **$INFORMIXDIR/msg**
7. In **$INFORMIXDIR/msg/english**

For more information about search paths for messages, see the description of **DBLANG** in the *IBM Informix GLS User's Guide*.

DBMONEY environment variable

Use the **DBMONEY** environment variable to specify the display format of values in columns of smallfloat, FLOAT, DECIMAL, or MONEY data types, and of complex data types derived from any of these data types.



$	is a currency symbol that precedes MONEY values in the default locale if no other *front* symbol is specified, or if **DBMONEY** is not set.
, *or* **.**	is a comma or period (the default) that separates the integral part from the fractional part of the FLOAT, DECIMAL, or MONEY value. Whichever symbol you do not specify becomes the thousands separator.
back	is a currency symbol that follows the MONEY value.
front	is a currency symbol that precedes the MONEY value.

The *back* symbol can be up to seven characters and can contain any character that the locale supports, except a digit, a comma (**,**), or a period (**.**) symbol. The *front* symbol can be up to seven characters and can contain any character that the locale supports except a digit, a comma (,), or a period (.) symbol. If you specify any character that is not a letter of the alphabet for *front* or *back*, you must enclose the *front* or *back* setting between single quotation (**'**) marks.

When you display MONEY values, IBM Informix products use the **DBMONEY** setting to format the output. **DBMONEY** has no effect, however, on the internal format of data values that are stored in columns of the database.

If you do not set **DBMONEY**, then MONEY values for the default locale, U.S. English, are formatted with a dollar sign ($) that precedes the MONEY value, a period (**.**) that separates the integral from the fractional part of the MONEY value, and no *back* symbol. For example, `100.50` is formatted as `$100.50`.

Suppose you want to represent MONEY values as DM (deutsche mark) units, using the currency symbol `DM` and comma (**,**) as the decimal separator. Enter the following command to set the **DBMONEY** environment variable:

```
setenv DBMONEY DM,
```

Here `DM` is the *front* currency symbol that precedes the MONEY value, and a comma separates the integral from the fractional part of the MONEY value. As a result, the value `100.50` is displayed as `DM100,50`.

For more information about how **DBMONEY** formats MONEY values in nondefault locales, see the *IBM Informix GLS User's Guide*.

DBONPLOAD environment variable

Use the **DBONPLOAD** environment variable to specify the name of the database that the **onpload** utility of the High Performance Loader (HPL) uses.

If **DBONPLOAD** is set, **onpload** uses the specified name as the name of the database; otherwise, the default name of the database is **onpload**.

```
►►──setenv──DBONPLOAD──dbname──────────────────────────────────────────────►◄
```

dbname
specifies the name of the database that the **onpload** utility uses.

For example, to specify the name **load_db** as the name of the database, enter the following command:

```
setenv DBONPLOAD load_db
```

For more information, see the *IBM Informix High-Performance Loader User's Guide*.

DBPATH environment variable

Use the **DBPATH** environment variable to identify the database servers that contain databases. DBPATH can also specify a list of directories (in addition to the current directory) in which DB-Access looks for command scripts (**.sql** files).

The CONNECT DATABASE, START DATABASE, and DROP DATABASE statements use **DBPATH** to locate the database under two conditions:

- If the location of a database is not explicitly stated
- If the database cannot be located in the default server

The CREATE DATABASE statement does not use **DBPATH**.

To add a new **DBPATH** entry to existing entries, see "Modifying an environment-variable setting" on page 3-4.

```
                      ┌─: [16]─────────────────────────────┐
                      ▼                                    │
►►──setenv──DBPATH──────┬─pathname──────────────────────┬──┴──────────────────►◄
                        ├─/ /──servername──/──full_pathname─┤
                        └─/ /──servername───────────────┘
```

full_pathname
is the full path, from **root**, of a directory where **.sql** files are stored.

pathname
is the valid relative path of a directory where **.sql** files are stored.

servername
is the name of a database server where databases are stored. You cannot reference database files with a *servername*.

DBPATH can contain up to 16 entries. Each entry must be less than 128 characters. In addition, the maximum length of **DBPATH** depends on the hardware platform on which you set **DBPATH**.

When you access a database with the CONNECT, DATABASE, START DATABASE, or DROP DATABASE statement, the search for the database is done first in the directory or database server specified in the statement. If no database server is specified, the default database server that was specified by the **INFORMIXSERVER** environment variable is used.

If the database is not located during the initial search, and if **DBPATH** is set, the database servers and directories in **DBPATH** are searched for in the specified database. These entries are searched in the same order in which they are listed in the **DBPATH** setting.

Using DBPATH with DB-Access

If you use DB-Access and select the **Choose** option from the **SQL** menu without having already selected a database, you see a list of all the **.sql** files in the directories listed in your **DBPATH**. After you select a database, the **DBPATH** is not used to find the **.sql** files. Only the **.sql** files in the current working directory are displayed.

Searching local directories

Use a pathname without a database server name to search for **.sql** scripts on your local computer. In the following example, the **DBPATH** setting causes DB-Access to search for the database files in your current directory and then in the Joachim and Sonja directories on the local computer:

```
setenv DBPATH /usr/joachim:/usr/sonja
```

As the previous example shows, if the pathname specifies a directory name but not a database server name, the directory is sought on the computer that runs the default database server that the **INFORMIXSERVER** specifies; see "INFORMIXSERVER environment variable" on page 3-54. For instance, with the previous example, if **INFORMIXSERVER** is set to **quality**, the **DBPATH** value is *interpreted,* as the following example shows, where the double slash precedes the database server name:

```
setenv DBPATH //quality/usr/joachim://quality/usr/sonja
```

Searching networked computers for databases

If you use more than one database server, you can set **DBPATH** explicitly to contain the database server and directory names that you want to search for databases. For example, if **INFORMIXSERVER** is set to **quality**, but you also want to search the **marketing** database server for **/usr/joachim**, set **DBPATH** as the following example shows:

```
setenv DBPATH //marketing/usr/joachim:/usr/sonja
```

Specifying a servername

You can set **DBPATH** to contain only database server names. This feature allows you to locate only databases; you cannot use it to locate command files.

The database administrator must include each database server mentioned by **DBPATH** in the **$INFORMIXDIR/etc/sqlhosts** file. For information about communication-configuration files and dbservernames, see your *IBM Informix Administrator's Guide* and the *IBM Informix Administrator's Reference*.

For example, if **INFORMIXSERVER** is set to **quality**, you can search for a database first on the **quality** database server and then on the **marketing** database server by setting **DBPATH,** as the following example shows:

```
setenv DBPATH //marketing
```

If you use DB-Access in this example, the names of all the databases on the **quality** and **marketing** database servers are displayed with the **Select** option of the DATABASE menu.

DBPRINT environment variable

Use the **DBPRINT** environment variable to specify the default printing program.

```
►►──setenv──DBPRINT──program─────────────────────────────────────────►◄
```

program
: Any command, shell script, or UNIX utility that produces standard ASCII output.

If you do not set **DBPRINT**, the default *program* is found in one of two places:

- For most BSD UNIX systems, the default program is **lpr**.
- For UNIX System V, the default program is usually **lp**.

Enter the following command to set the **DBPRINT** environment variable to specify **myprint** as the print program:

```
setenv DBPRINT myprint
```

DBREMOTECMD environment variable (UNIX)

Use the **DBREMOTECMD** environment variable to override the default remote shell to perform remote tape operations with the database server.

You can set **DBREMOTECMD** to a simple command or to a full path name.

```
►►──setenv──DBREMOTECMD──┬─command──┬────────────────────────────────►◄
                         └─pathname─┘
```

command
: A command to override the default remote shell.

pathname
: A path name to override the default remote shell.

If you do not specify the full path name, the database server searches your **PATH** for the specified *command*. You should use the full path name syntax on interactive UNIX platforms to avoid problems with similarly named programs in other directories and possible confusion with the *restricted shell* (`/usr/bin/rsh`).

The following command sets **DBREMOTECMD** for a simple command name:

```
setenv DBREMOTECMD rcmd
```

The next command to set **DBREMOTECMD** specifies a full path name:

```
setenv DBREMOTECMD /usr/bin/remsh
```

For more information about using remote tape devices for backups, see Specify a remote device.

DBSPACETEMP environment variable

The **DBSPACETEMP** environment variable specifies the dbspaces in which temporary tables are built. The list can include standard dbspaces, temporary dbspaces, or both.

```
                        .-,-------.
                        V         |
>>--setenv--DBSPACETEMP----dbspace-+---------------------------------><
```

dbspace
is the name of an existing standard or temporary dbspace.

You can list dbspaces, separated by colon (:) or comma (,) symbols, to designate space for temporary tables across physical storage devices. For example, the following command to set the **DBSPACETEMP** environment variable specifies three dbspaces for temporary tables:

```
setenv DBSPACETEMP sorttmp1:sorttmp2:sorttmp3
```

DBSPACETEMP overrides any default dbspaces that the DBSPACETEMP parameter specifies in the configuration file of the database server. For UPDATE STATISTICS operations, DBSPACETEMP is used only when you specify the HIGH keyword option.

On UNIX platforms, you might have better performance if the list of dbspaces in **DBSPACETEMP** is composed of chunks that are allocated as raw devices.

The number of dbspaces is limited by the maximum size of the environment variable, as defined by your operating system. Your database server does not create a dbspace specified by the environment variable if the dbspace does not exist.

The two classes of temporary tables are *explicit* temporary tables that the user creates and *implicit* temporary tables that the database server creates. Use **DBSPACETEMP** to specify the dbspaces for both types of temporary tables.

If you create an explicit temporary table with the CREATE TEMP TABLE statement and do not specify a dbspace for the table either in the IN *dbspace* clause or in the FRAGMENT BY clause, the database server uses the settings in **DBSPACETEMP** to determine where to create the table.

If you create an explicit temporary table with the SELECT INTO TEMP statement, the database server uses the settings in **DBSPACETEMP** to determine where to create the table.

If **DBSPACETEMP** is set, and the dbspaces that it lists include both logging and non-logging dbspaces, the database server stores temporary tables that implicitly or explicitly support transaction logging in a logged dbspace, and non-logging temporary tables in a non-logging dbspace.

The database server creates implicit temporary tables for its own use while executing join operations, SELECT statements with the GROUP BY clause, SELECT statements with the ORDER BY clause, and index builds.

When it creates explicit or implicit temporary tables, the database server uses disk space for writing the temporary data. If there are conflicts among settings or statement specifications for the location of a temporary table, these conflicts are resolved in this descending (highest to lowest) order of precedence:

1. On UNIX platforms, the operating-system directory or directories that the environment variable **PSORT_DBTEMP** specifies, if this is set
2. The dbspace or dbspaces that the environment variable **DBSPACETEMP** specifies, if this is set

3. The dbspace or dbspaces that the ONCONFIG parameter DBSPACETEMP specifies.
4. The operating-system file space specified by the DUMPDIR configuration parameter
5. The directory $INFORMIXDIR/tmp (UNIX) or $INFORMIXDIR\tmp (Windows).

Important: If the **DBSPACETEMP** environment variable is set to an invalid value, the database server defaults to the root dbspace for explicit temporary tables and to **/tmp** for implicit temporary tables, rather than to the setting of the DBSPACETEMP configuration parameter. In this situation, the database server might fill **/tmp** to the limit and eventually bring down the database server or kill the file system.

DBTEMP environment variable

The **DBTEMP** environment variable is used by DB-Accessand IBM Informix Enterprise Gateway products and by IBM Informix and by earlier database servers. **DBTEMP** resembles **DBSPACETEMP**, specifying the directory in which to place temporary files and temporary tables.

```
►►──setenv──DBTEMP──pathname─────────────────────────────────────────────►◄
```

pathname
The full path name of the directory for temporary files and tables.

For DB-Access to work correctly on Windows platforms, **DBTEMP** should be set to `$INFORMIXDIR/infxtmp`.

The following example sets **DBTEMP** to the path name `usr/magda/mytemp` for UNIX systems that use the C shell:

```
setenv DBTEMP usr/magda/mytemp
```

Important: **DBTEMP** can point to an NFS-mounted directory only if the vendor of that NFS device is certified by IBM.

If **DBTEMP** is not set, the database server creates temporary files in the `/tmp` directory and temporary tables in the **DBSPACETEMP** directory. See “DBSPACETEMP environment variable” on page 3-30 for the default if **DBSPACETEMP** is not set. Similarly, if you do not set **DBTEMP** on the client system, temporary files (such as those created for scroll cursors) are created in the `/tmp` directory.

You might experience unexpected behavior or failure in operations on values of large or complex data types, such as BYTE or ROW, if DBTEMP is not set.

DBTIME environment variable

The **DBTIME** environment variable specifies a formatting mask for the display and data-entry format of DATETIME values.

The**DBTIME** environment variable is useful in contexts where the DATETIME data values to be formatted by **DBTIME** have the same precision as the specified **DBTIME** setting. You might encounter unexpected or invalid display formats for DATETIME values that are declared with a different DATETIME qualifier.

```
                           ┌───────────────────────────────────────────────────┐
                           ▼                                                   │
►►──setenv──DBTIME──'──────┬─literal───────────────────────────────────────────┬┴──'──────►◄
                           └─%──┬─────┬──┬─────┬──┬───────────────┬─special─┘
                                ├─-───┤  └─min─┘  └─.──precision──┘
                                └─0───┘
```

literal is a literal white space or any printable character.

min is a literal integer, setting the minimum number of characters in the substring for the value that *special* specifies.

precision
is the number of digits for the value of any time unit, or the maximum number of characters in the name of a month.

special is one of the placeholder characters that are listed following.

These terms and symbols are described in the pages that follow.

This quoted string can include literal characters and placeholders for the values of individual time units and other elements of a DATETIME value. **DBTIME** takes effect only when you call certain IBM Informix ESQL/C DATETIME routines. (For details, see the *IBM Informix ESQL/C Programmer's Manual*.) If **DBTIME** is not set, the behavior of these routines is undefined, and `"YYYY-MM-DD hh:mm:ss.fffff"` is the default display and input format for DATETIME YEAR TO FRACTION(5) literal values in the default locale.

The percentage (%) symbol gives special significance to the *special* placeholder symbol that follows. Without a preceding % symbol, any character within the formatting mask is interpreted as a literal character, even if it is the same character as one of the placeholder characters in the following list. Note also that the *special* placeholder symbols are case sensitive.

The following characters within a **DBTIME** format string are placeholders for time units (or for other features) within a DATETIME value.

%b is replaced by the abbreviated month name.

%B is replaced by the full month name.

%d is replaced by the day of the month as a decimal number [01,31].

%F*n* is replaced by a fraction of a second with a scale that the integer *n* specifies. The default value of *n* is 2; the range of *n* is $0 \le n \le 5$.

%H is replaced by the hour (24-hour clock).

%I is replaced by the hour (12-hour clock).

%M is replaced by the minute as a decimal number [00,59].

%m is replaced by the month as a decimal number [01,12].

%p is replaced by A.M. or P.M. (or the equivalent in the locale file).

%S is replaced by the second as a decimal number [00,59].

%y is replaced by the year as a four-digit decimal number.

%Y is replaced by the year as a four-digit decimal number. User must enter a four-digit value.

%% is replaced by `%` (to allow a literal `%` character in the format string).

For example, consider this display format for DATETIME YEAR TO SECOND:

```
Mar 21, 2013 at 16 h 30 m 28 s
```

If the user enters a two-digit year value, this value is expanded to 4 digits according to the **DBCENTURY** environment variable setting. If **DBCENTURY** is not set, then the string 19 is used by default for the first two digits.

Set **DBTIME** as the following command line (for the C shell) shows:

```
setenv DBTIME '%b %d, %Y at %H h %M m %S s'
```

The default **DBTIME** produces the following ANSI SQL string format:

```
2001-03-21 16:30:28
```

You can set the default **DBTIME** as the following example shows:

```
setenv DBTIME '%Y-%m-%d %H:%M:%S'
```

An optional field width and precision specification (*w.p*) can immediately follow the percent (%) character. It is interpreted as follows:

w	Specifies the minimum field width. The value is right-justified with blank spaces on the left.
-*w*	Specifies the minimum field width. The value is left-justified with blank spaces on the right.
0*w*	Specifies the minimum field width. The value is right-justified and padded with zeros on the left.
p	Specifies the precision of `d`, `H`, `I`, `m`, `M`, `S`, `y`, and `Y` time unit values, or the maximum number of characters in `b` and `B` month names.

The following limitations apply to field-width and precision specifications:

- If the data value supplies fewer digits than *precision* specifies, the value is padded with leading zeros.
- If a data value supplies more characters than *precision* specifies, excess characters are truncated from the right.
- If no field width or precision is specified for `d`, `H`, `I`, `m`, `M`, `S`, or `y` placeholders, `0.2` is the default, or `0.4` for the `Y` placeholder.
- A *precision* specification is significant only when converting a DATETIME value to an ASCII string, but not vice versa.

The `F` placeholder does not support this field-width and precision syntax.

Important: Any separator character between the `%S` and `%F` directives for DATETIME user formats must be explicitly defined. Specifying `%S%F` concatenates the digits that represent the integer and fractional parts of the seconds value.

Like **DBDATE**, **GL_DATE**, or **GL_DATETIME**, or **USE_DTENV**, the **DBTIME** setting controls only the character-string representation of data values. It cannot change the internal storage format of the DATETIME column. (For additional information about formatting DATE values, see the discussion of **DBDATE** in the topic "DBDATE environment variable" on page 3-22.)

DBTIME formats in nondefault locales

If you specify a locale other than U.S. English, the locale defines the culture-specific display formats for DATETIME values. To change the default display format, change the setting of **DBTIME**, or of the **GL_DATETIME** and **USE_DTENV** environment variables.

In East Asian locales that support era-based dates, **DBTIME** can also specify Japanese or Taiwanese eras. See *IBM Informix GLS User's Guide* for details of additional placeholder symbols for setting **DBTIME** to display era-based DATETIME values, and for descriptions of the **GL_DATETIME**, **GL_DATE**, and **USE_DTENV** environment variables.

Related reference:

"DATETIME data type" on page 2-12

DBUPSPACE environment variable

Use the **DBUPSPACE** environment variable to specify the amount of system disk space and the amount of memory that the UPDATE STATISTICS MEDIUM and UPDATE STATISTICS HIGH statement can use when it reads and sorts column values to construct column distributions. The **DBUPSPACE** setting can also request SET EXPLAIN output to describe the execution path for calculating the statistical distributions.

```
                            ┌─1024─┐  ┌─ :──15────────┐
►►──setenv──DBUPSPACE───────┴─disk─┴──┴─ :── memory───┴──┬──────────────────┬────────────►◄
                                                         └─ :── directive───┘
```

disk is an unsigned integer, specifying the disk space (in KiB) to allocate for sorting in UPDATE STATISTICS MEDIUM and HIGH operations.

memory
is an unsigned integer, specifying the maximum amount of sorting memory (in MiB, in the range from 4 to 50 megabytes) to allocate without using PDQ.

directive
is an unsigned integer, encoding one of the following directives for the UPDATE STATISTICS execution plan:

- 1: Do not use any indexes for sorting. Print the entire plan for update statistics in the `sqexplain.out` file.
- 2: Do not use any indexes for sorting. Do not print the plan for update statistics.
- 3 or greater: Use available indexes for sorting. Print the entire plan for update statistics in explain output file.

For example, to set **DBUPSPACE** to 2,500 KiB of disk space and 1 megabyte of memory, enter this command:

```
setenv DBUPSPACE 2500:1
```

After you set this value, the database server will attempt to use no more than 2,500 KiB of disk space during the execution of an UPDATE STATISTICS MEDIUM or HIGH statement. If a table requires 5 megabytes of disk space for sorting, then UPDATE STATISTICS accomplishes the task in two passes; the distributions for one half of the columns are constructed with each pass. For a table of a given storage size, this parameter determines the number of passes, but no pass can write less than a full column.

If you do not set **DBUPSPACE**, the default setting is 1 megabyte (1,024 KiB) for *disk*, and 15 megabytes for *memory*. If you attempt to set the first **DBUPSPACE** parameter to any value less than 1,024 KiB, it is automatically set to 1,024 KiB, but no error message is returned. If this *disk* value is not large enough to allow more than one distribution to be constructed at a time, at least one distribution is done, even if the amount of disk space required to do this is more than what **DBUPSPACE** specifies. That is, regardless of the *disk* parameter setting for **DBUPSPACE**, the largest individual column storage requirement of a table determines the actual upper limit on disk space for a single pass in any UPDATE STATISTICS HIGH or MEDIUM operation.

Related information:

Default name and location of the explain output file on UNIX

Default name and location of the output file on Windows

DEFAULT_ATTACH environment variable

The **DEFAULT_ATTACH** environment variable supports the legacy behavior of Version 7.x of IBM Informix, in which the pages of nonfragmented B-tree indexes on nonfragmented tables were stored, by default, in the same dbspace partition as the data pages. (The name "**DEFAULT_ATTACH**" derives from an obsolete definition of an *attached index*, a term that now refers to an index whose fragmentation strategy is the same as the fragmentation strategy of its table. Do not confuse the obsolete Version 7.x definition with this current definition.)

```
►►──setenv──DEFAULT_ATTACH──1──────────────────────────────────────────►◄
```

If the **DEFAULT_ATTACH** environment variable is set to 1, then by default, the pages of nonfragmented B-tree indexes on nonfragmented tables are stored in the same partition (and in the same dbspace) that stores data pages of the table. The IN TABLE keywords of the CREATE INDEX statement are not required (but do not return an error).

Setting **DEFAULT_ATTACH** to 1 has no effect, however, on any other types of indexes, whose pages are always stored in separate partitions from the data pages of the indexed table. These index types whose storage distribution is always different from that of their table include

- R-tree indexes,
- functional indexes,
- forest of trees indexes,
- fragmented indexes,
- and indexes on fragmented tables.

Index storage in the same partition as the data pages is supported only for nonfragmented B-tree indexes on nonfragmented tables.

If **DEFAULT_ATTACH** is not set, then by default, any CREATE INDEX statement that does not specify IN TABLE as its Storage Options clause creates an index whose pages are stored in partitions separate from the data pages. This release of IBM Informix can support existing indexes that were created by Version 7.x of IBM Informix.

Important: Future releases of IBM Informix might not continue to support **DEFAULT_ATTACH.** Developing new applications that depend on this deprecated feature is not recommended.

DELIMIDENT environment variable

The **DELIMIDENT** environment variable specifies that strings enclosed between double quotation (") marks are delimited database identifiers.

The **DELIMIDENT** environment variable is also supported on client systems, where it can be set to y, to n, or to no setting.

- **y** specifies that client applications must use single quotation (') symbols to delimit character strings, and must use double quotation (") symbols only around delimited SQL identifiers, which can support a larger character set than is valid in undelimited identifiers. Letters within delimited strings or delimited identifiers are case-sensitive. This is the default value for OLE DB and .NET.
- **n** specifies that client applications can use double quotation (") or single quotation (') symbols to delimit character strings, but not to delimit SQL identifiers. If the database server encounters a string delimited by double or single quotation symbols in a context where an SQL identifier is required, it issues an error. An owner name that qualifies an SQL identifier can be delimited by single quotation (') symbols. You must use a pair of the same quotation symbols to delimit a character string.

  This is the default value for ESQL/C, JDBC, and ODBC. APIs that have ESQL/C as an underlying layer, such as IBM Informix 4GL, the DataBlade API (LIBDMI), and the C++ API, behave as ESQL/C, and use 'n' as the default if no value for DELIMIDENT is specified on the client system.
- Specifying the DELIMIDENT environment variable with no value on the client system requires client applications to use the DELIMIDENT setting that is the default for their application programming interface (API).

```
►►──setenv──DELIMIDENT─────────────────────────────────────────────────►◄
```

No value is required; **DELIMIDENT** takes effect if it exists, and it remains in effect while it is on the list of environment variables. Removing DELIMIDENT when it is set at the server level requires restarting the server.

Delimited identifiers can include white space (such as the phrase "**Vitamin E**") or can be identical to SQL keywords, (such as "**TABLE**" or "**USAGE**"). You can also use them to declare database identifiers that contain characters outside the default character set for SQL identifiers (such as "**Column #6**"). In the default locale, this set consists of letters, digits, and the underscore (_) symbol.

Even if DELIMIDENT is set, you can use single quotation (') symbols to delimit authorization identifiers as the owner name component of a database object name, as in the following example:

```
RENAME COLUMN 'Owner'.table2.collum3 TO column3;
```

This example is an exception to the general rule that when **DELIMIDENT** is set, the SQL parser interprets character strings delimited by single quotation symbols as string literals, and interprets character strings delimited by double quotation symbols (") as SQL identifiers.

Database identifiers (also called *SQL identifiers*) are names for database objects, such as tables and columns. *Storage identifiers* are names for storage objects, such as dbspaces, blobspaces, and sbspaces. You cannot use **DELIMIDENT** to declare storage identifiers that contain characters outside the default SQL character set.

Delimited identifiers are case sensitive. To use delimited identifiers, applications in Informix ESQL/C must set **DELIMIDENT** at compile time and at run time.

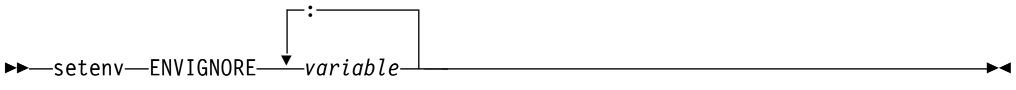
Important: If **DELIMIDENT** is not already set, you should be aware that setting it can cause the failure of existing `.sql` scripts or client applications that use double (") quotation marks in contexts other than delimiting SQL identifiers, such as delimiters of string literals. You must use single (') rather than double quotation marks for delimited constructs that are not SQL identifiers if **DELIMIDENT** is set.

On UNIX systems that use the C shell and on which **DELIMIDENT** has been set, you can disable this feature (which causes anything between double quotation symbols to be interpreted as an SQL identifier) by the command:

```
unsetenv  DELIMIDENT
```

ENVIGNORE environment variable (UNIX)

The **ENVIGNORE** environment variable can deactivate specified environment variable settings in the common (shared) configuration file, `informix.rc`, and private environment-configuration file, `.informix`.



variable
: The name of an environment variable to be deactivated.

Use colon (:) symbols between consecutive *variable* names. For example, to ignore the **DBPATH** and **DBMONEY** entries in the environment-configuration files, enter the following command:

```
setenv ENVIGNORE DBPATH:DBMONEY
```

The common environment-configuration file is stored in `$INFORMIXDIR/etc/informix.rc`.

The private environment-configuration file is stored in the home directory of the user as `.informix`.

For information about creating or modifying an environment-configuration file, see "Setting environment variables in a configuration file" on page 3-2.

ENVIGNORE itself cannot be set in an environment-configuration file.

FET_BUF_SIZE environment variable

The **FET_BUF_SIZE** environment variable can override the default setting for the size of the fetch buffer for all data types except BYTE and TEXT values. For ANSI databases, you must set transactions to READ ONLY for the **FET_BUF_SIZE**environment variable to improve performance, otherwise rows are returned one by one.

```
►►──setenv──FET_BUF_SIZE──size──────────────────────────────────────►◄
```

size	is a positive integer that is larger than the default buffer size, but no greater than 2147483648 (2GB), specifying the size (in bytes) of the fetch buffer that holds data retrieved by a query.

For example, to set a buffer size to `5,000` bytes on a UNIX system that uses the C shell, set **FET_BUF_SIZE** by entering the following command:

```
setenv FET_BUF_SIZE 5000
```

When **FET_BUF_SIZE** is set to a valid value, the new value overrides the default value (or any previously set value of **FET_BUF_SIZE**). The default setting for the fetch buffer is dependent on row size.

The processing of BYTE and TEXT values is not affected by **FET_BUF_SIZE**.

No error is raised if **FET_BUF_SIZE** is set to a value that is less than the default size or is larger than 2147483648 (2GB). In these cases, however, the invalid fetch buffer size is ignored, and the default size is in effect.

A valid **FET_BUF_SIZE** setting is in effect for the local database server and for any remote database server from which you retrieve rows through a distributed query in which the local server is the coordinator and the remote database is subordinate. The greater the size of the buffer, the more rows can be returned, and the less frequently the client application must wait while the database server returns rows. A large buffer can improve performance by reducing the overhead of filling the client-side buffer.

IFMXMONGOAUTH environment variable

Set the **IFMXMONGOAUTH** environment variable to enable PAM authentication for MongoDB clients through the wire listener.

You can set the **IFMXMONGOAUTH** environment variable to any value or to no value.

```
►►──setenv──IFMXMONGOAUTH──1────────────────────────────────────────►◄
```

Setting the **IFMXMONGOAUTH** environment variable is a prerequisite to configuring PAM authentication for MongoDB clients.

You can disable the **IFMXMONGOAUTH** environment variable with this command:

```
unsetenv IFMXMONGOAUTH
```

Related information:

Configuring PAM authentication

IFX_DEF_TABLE_LOCKMODE environment variable

The **IFX_DEF_TABLE_LOCKMODE** environment variable can specify the default lock mode for database tables that are subsequently created without explicitly specifying the LOCKMODE PAGE or LOCKMODE ROW keywords. This feature is convenient if you must create several tables of the same lock mode. UNIX systems that use the C shell support the following syntax:

```
►►─setenv─IFX_DEF_TABLE_LOCKMODE──┬─PAGE─┬──────────────────────────────►◄
                                  └─ROW──┘
```

PAGE The default lock mode is page-level granularity. This value disables the LAST COMMITTED feature of COMMITTED READ.

ROW The default lock mode is row-level granularity.

Similar functionality is available by setting the DEF_TABLE_LOCKMODE parameter of the ONCONFIG file to PAGE or ROW. When a table is created or modified, any conflicting lock mode specifications are resolved according to the following descending (highest to lowest) order of precedence:

1. Explicit LOCKMODE specification of CREATE TABLE or ALTER TABLE
2. **IFX_DEF_TABLE_LOCKMODE** environment variable setting
3. DEF_TABLE_LOCKMODE parameter setting in the ONCONFIG file
4. The system default lock mode (= page mode)

To make the DEF_TABLE_LOCKMODE setting the default mode (or to restore the system default if DEF_TABLE_LOCKMODE is not set) use the command:

```
unsetenv IFX_DEF_TABLE_LOCKMODE
```

If **IFX_DEF_TABLE_LOCKMODE** is set in the environment of the database server before running **oninit**, then its scope is all sessions of the database server (just as if DEF_TABLE_LOCKMODE were set in the ONCONFIG file). If **IFX_DEF_TABLE_LOCKMODE** is set in the shell, or in the **$HOME/.informix** or **$INFORMIXDIR/etc/informix.rc** files, then the scope is restricted to the current session (if you set it in the shell) or to the individual user.

Important: This has no effect on existing tables. If you specify *ROW* as the lock mode, the database will use this to restore, recover, or copy data. For tables that were created in *PAGE* mode, this might cause lock-table overflow or performance degradation.

IFX_DIRECTIVES environment variable

The **IFX_DIRECTIVES** environment variable setting determines whether the optimizer allows query optimization directives from within a query. The **IFX_DIRECTIVES** environment variable is set on the client.

You can specify either `ON` and `OFF` or `1` and `0` to set the environment variable.

```
►►─setenv─IFX_DIRECTIVES──┬─1─┬─────────────────────────────────────────►◄
                          └─0─┘
```

1 Optimizer directives accepted

0 Optimizer directives not accepted

The setting of the **IFX_DIRECTIVES** environment variable overrides the value of the DIRECTIVES configuration parameter that is set for the database server. If the **IFX_DIRECTIVES** environment variable is not set, however, then all client sessions will inherit the database server configuration for directives that the ONCONFIG parameter DIRECTIVES determines. The default setting for the **IFX_DIRECTIVES** environment variable is `ON`.

For more information about the DIRECTIVES parameter, see the *IBM Informix Administrator's Reference*. For more information about the performance impact of directives, see your *IBM Informix Performance Guide*.

IFX_EXTDIRECTIVES environment variable

The **`IFX_EXTDIRECTIVES`** environment variable specifies whether the query optimizer allows external query optimization directives from the **sysdirectives** system catalog table to be applied to queries in existing applications.

You have two options for setting the **`IFX_EXTDIRECTIVES`** environment variable:

- Global, for all users:

  On the server, set **`IFX_EXTDIRECTIVES`** in the environment as user `informix` and then run the **oninit** command.
- Client specific:

  On the client, set **`IFX_EXTDIRECTIVES`** in the environment. When **`IFX_EXTDIRECTIVES`** is set in the client environment, the client setting are used regardless of the server (global) setting.

You can determine the server setting using the **onstat -g env** command.

You can specify either `ON` and `OFF` or `1` and `0` to set the environment variable.

```
►►──setenv──IFX_DIRECTIVES──┬─1─┬──────────────────────────────────►◄
                            └─0─┘
```

1 External optimizer directives accepted

0 External optimizer directives not accepted

Queries within a given client application can use external directives if both the EXT_DIRECTIVES parameter in the configuration file of the database server and the **IFX_EXTDIRECTIVES** environment variable setting on the client system are both set to 1 or ON. If **IFX_EXTDIRECTIVES** is not set, external directives are supported only if the ONCONFIG parameter EXT_DIRECTIVES is set to 2. The following table summarizes the effect of valid IFX_EXTDIRECTIVES and EXT_DIRECTIVES settings on support for external optimizer directives.

Table 3-3. Effect of IFX_EXTDIRECTIVES and EXT_DIRECTIVES settings on external directives

	EXT_DIRECTIVES = 0	**EXT_DIRECTIVES = 1**	**EXT_DIRECTIVES = 2**
IFX_EXTDIRECTIVES No setting	OFF	OFF	ON
IFX_EXTDIRECTIVES0 = OFF	OFF	OFF	OFF

Table 3-3. Effect of IFX_EXTDIRECTIVES and EXT_DIRECTIVES settings on external directives (continued)

	EXT_DIRECTIVES = 0	**EXT_DIRECTIVES = 1**	**EXT_DIRECTIVES = 2**
IFX_EXTDIRECTIVES1 = ON	OFF	ON	ON

The database server interprets any EXT_DIRECTIVES setting besides 1 or 2 (or no setting) as equivalent to OFF, disabling support for external directives. Any value of **IFX_EXTDIRECTIVES** other than 1 has the same effect for the client.

For information about how to define external optimizer directives, see the description of the SAVE EXTERNAL DIRECTIVES statement of SQL in the *IBM Informix Guide to SQL: Syntax*. For more information about the EXT_DIRECTIVES configuration parameter, see the *IBM Informix Administrator's Reference*. For more information about the performance impact of directives, see your *IBM Informix Performance Guide*.

IFX_LARGE_PAGES environment variable

The **IFX_LARGE_PAGES** environment variable specifies whether the database server can use large pages on platforms where the hardware and the operating system support large pages of shared memory. If this is enabled in the server environment, IBM Informix can use the large pages for non-message shared memory segments that are located in physical memory.

The **IFX_LARGE_PAGES** environment variable is supported only on AIX, Solaris, and Linux operating systems. The setting of **IFX_LARGE_PAGES** has no effect on Informix if the operating system does not support large pages, or if large pages are not configured on the system.

You can specify either `1` or `0` to set this environment variable.

```
►►──setenv──IFX_LARGE_PAGES──┬─1─┬──────────────────────────────────────►◄
                             └─0─┘
```

0 The use of large pages is disabled. This is the default on AIX systems.

1 The use of large pages is enabled. This is the default on Solaris and Linux systems.

The DBSA must use operating system commands to configure the large pages. See the operating system documentation for the configuration procedures.

Informix can use large pages for non-message shared memory segments that are locked in physical memory, if sufficient large pages are configured and available. The RESIDENT configuration parameter controls whether a shared memory segment is locked in physical memory, so that the segment cannot be swapped. If there are insufficient large pages to hold a segment, the segment might contain a mixture of large pages and regular pages.

On AIX the large pages used by Informix are 16 MB in size.

On Linux x86_64 the large pages used by Informix are defined by the `Hugepagesize` entry in the `/proc/meminfo` file.

Informix aligns the segment address and rounds up to the segment size automatically. In addition to messages regarding rounding, the server prints an informational message to the server log file whenever it attempts to use large pages to store a segment.

When **IFX_LARGE_PAGES** is enabled, the use of large pages can offer significant performance benefits in large memory configurations.

Related information:

RESIDENT configuration parameter

IFX_LOB_XFERSIZE environment variable

Use the **IFX_LOB_XFERSIZE** environment variable to specify the number of bytes in a CLOB or BLOB data type to transfer from a client application to the database server before checking whether an error has occurred.

The error check occurs each time the specified number of bytes is transferred. If an error occurs, the remaining data is not sent and an error is reported. If no error occurs, the file transfer will continue until it finishes.

For example, if the value of **IFX_LOB_XFERSIZE** is set to 10485760 (10 MB), then error checking will occur after every 10485760 bytes of the CLOB or BLOB data is sent. If **IFX_LOB_XFERSIZE** is not set, the error check occurs after the entire BLOB or CLOB data is transferred.

The valid range for **IFX_LOB_XFERSIZE** is from 1 to 9223372036854775808 bytes. The **IFX_LOB_XFERSIZE** environment variable is set on the client.

```
►►──setenv──IFX_LOB_XFERSIZE──value────────────────────────────────►◄
```

value the number of bytes in a CLOB or BLOB to transfer from a client application to the database server before checking whether an error has occurred

You should adjust the value of **IFX_LOB_XFERSIZE** to suit your environment. Set **IFX_LOB_XFERSIZE** low enough so that transmission errors of large BLOB or CLOB data types are detected early, but not so low that excessive network resources are used.

IFX_LONGID environment variable

The **IFX_LONGID** environment variable setting and the version number of the client application determine whether a given client application is capable of handling long identifiers. (Older versions of IBM Informix restricted SQL identifiers to 18 or fewer bytes; *long identifiers* can have up to 128 bytes when **IFX_LONGID** is set.) Valid **IFX_LONGID** values are 1 and 0.

```
►►──setenv──IFX_LONGID──┬─1─┬──────────────────────────────────────►◄
                        └─0─┘
```

1 Client supports long identifiers.

0 Client cannot support long identifiers.

When **IFX_LONGID** is set to zero, applications display only the first 18 bytes of long identifiers, without indicating (by +) that truncation has occurred.

If **IFX_LONGID** is unset or is set to a value other than 1 or 0, the determination is based on the internal version of the client application. If the (server-based) version is not less than 9.0304, or is in the (CSDK-based) range 2.90 ≤ *version* < 4.0, the client is considered capable of handling long identifiers. Otherwise, the client application is considered incapable.

The **IFX_LONGID** setting overrides the internal version of the client application. If the client cannot handle long identifiers despite a newer version number, set **IFX_LONGID** to 0. If the client version can handle long identifiers despite an older version number, set **IFX_LONGID** to 1.

If you set **IFX_LONGID** on the client, the setting affects only that client. If you start the database server with **IFX_LONGID** set, all client applications use that setting by default. If **IFX_LONGID** is set to different values on the client and on the database server, however, the client setting takes precedence.

Important: ESQL executables that have been built with the `-static` option using the `libos.a` library version that does not support long identifiers cannot use the **IFX_LONGID** environment variable. You must recompile such applications with the new `libos.a` library that includes support for long identifiers. Executables that use shared libraries (no `-static` option) can use **IFX_LONGID** without recompilation provided that they use the new `libifos.so` that provides support for long identifiers. For details, see your ESQL product publication.

IFX_NETBUF_PVTPOOL_SIZE environment variable (UNIX)

Use the **IFX_NETBUF_PVTPOOL_SIZE** environment variable to specify the maximum size of the free (unused) private network buffer pool for each database server session.

```
►►──setenv──IFX_NETBUF_PVTPOOL_SIZE──count──────────────────────────────►◄
```

count an integer specifying the number of units (buffers) in the pool.

The default size is 1 buffer. If **IFX_NETBUF_PVTPOOL_SIZE** is set to 0, then each session obtains buffers from the free global network buffer pool. You must specify the value in decimal form.

IFX_NETBUF_SIZE environment variable

Use the **IFX_NETBUF_SIZE** environment variable to configure the network buffers to the optimum size. This environment variable specifies the size of all network buffers in the free (unused) global pool and the private network buffer pool for each database server session.

```
►►──setenv──IFX_NETBUF_SIZE──size───────────────────────────────────────►◄
```

size is the integer size (in bytes) for one network buffer.

The default size is 4 KB (4,096 bytes). The maximum size is 64 KB (65,536 bytes) and the minimum size is 512 bytes. You can specify the value in hexadecimal or decimal form.

Tip: You cannot set a different size for each session.

IFX_NO_SECURITY_CHECK environment variable (UNIX)

The **IFX_NO_SECURITY_CHECK** environment variable allows user **informix** or **root** to complete operations with a database server instance even when the IBM Informix utilities detect that the `$INFORMIXDIR` path is not secure. Do not use this environment variable unless your system setup makes it absolutely necessary to do so.

The purpose of **IFX_NO_SECURITY_CHECK** is for environments where the database server started but while running it detects that the runtime path is not secure anymore. In this case, a superuser might be required to stop the database server in order to remedy the security flaw. With this environment variable, either user **informix** or **root** can use the **onmode** utility to shut down a nonsecure Informix instance, which would otherwise not be possible because key programs do not run when the `$INFORMIXDIR` path is not secure.

There is some risk in using this environment variable, but in some circumstances it might be necessary to remedy a bigger security problem. The requirement that only user **informix** or **root** can invoke **IFX_NO_SECURITY_CHECK** makes it unlikely that an illegitimate user would be able to run it.

To use this environment variable, set it to any non-empty string.

```
►►──setenv──IFX_NO_SECURITY CHECK──1────────────────────────────────────►◄
```

1 Any value entered here when running this environment variable disables the **onsecurity** utility.

Important: Turn off this environment variable after you have finished troubleshooting the security problem.

IFX_NO_TIMELIMIT_WARNING environment variable

Trial or evaluation versions of IBM Informix software products, which cease to function when some time limit has elapsed since the software was installed, by default issue warning messages that tell users when the license will expire. If you set the **IFX_NO_TIMELIMIT_WARNING** environment variable, however, the time-limited software does not issue these warning messages.

```
►►──setenv──IFX_NO_TIMELIMIT_WARNING────────────────────────────────────►◄
```

For users who dislike viewing warning messages, this feature is an alternative to redirecting the error output. Setting **IFX_NO_TIMELIMIT_WARNING** has no effect, however, on when a time-limited license expires; the software ceases to function at the same point in time when it would if this environment variable had not been set. If you do set **IFX_NO_TIMELIMIT_WARNING**, users will not see potentially annoying warnings about the impending license expiration, but some users might be annoyed at you when the database server (or whatever software has a time-limited license) ceases to function without any warning.

IFX_NODBPROC environment variable

The **IFX_NODBPROC** environment variable lets you prevent the database server from running the sysdbopen( ) or sysdbclose( ) procedure. These procedures cannot be run if this environment variable is set to any value.

```
►►—setenv—IFX_NODBPROC—string——————————————————————————————►◄
```

string Any value prevents the database server from running sysdbopen( ) or sysdblcose( ).

IFX_NOT_STRICT_THOUS_SEP environment variable

IBM Informix requires the thousands separator to have 3 digits following it. For example, 1,000 is considered correct, and 1,00 is considered wrong. In previous releases, both formats were considered correct.

```
►►—setenv—IFX_NOT_STRICT_THOUS_SEP—n————————————————————————►◄
```

n Set *n* to 1 for the behavior in previous releases, which is that the thousands separator can have fewer than three digits following it.

IFX_ONTAPE_FILE_PREFIX environment variable

When TAPEDEV and LTAPEDEV specify directories, use the **IFX_ONTAPE_FILE_PREFIX** environment variable to specify a prefix for backup file names that replaces the *hostname_servernum* format. If no value is set, file names are *hostname_servernum*_L*n* for levels and *hostname_servernum*_Log*nnnnnnnnnn* for log files.

If you set the value of IFX_ONTAPE_FILE_PREFIX to My_Backup, the backup file names have the following names:

- My_Backup_L0
- My_Backup_L1
- My_Backup_L2
- My_Backup_Log0000000001
- My_Backup_Log0000000002

```
►►—setenv—IFX_ONTAPE_FILE_PREFIX—string—————————————————————►◄
```

string The prefix to use for the names of backup files.

IFX_PAD_VARCHAR environment variable

The **IFX_PAD_VARCHAR** environment variable setting controls how the database server sends and receives VARCHAR and NVARCHAR data values. Valid **IFX_PAD_VARCHAR** values are 1 and 0.

```
►►—setenv—IFX_PAD_VARCHAR——┬─1─┬————————————————————————————►◄
                           └─0─┘
```

1 Transmit the entire structure, up to the declared *max* size.

0 Transmit only the portion of the structure containing data.

For example, to send the string "ABC" from a column declared as NVARCHAR(255) when **IFX_PAD_VARCHAR** is set to 0 would send 3 bytes.

If the setting were 1 in the previous example, however, the number of bytes sent would be 255 bytes.

The effect **IFX_PAD_VARCHAR** is context-sensitive. In a low-bandwidth network, a setting of 0 might improve performance by reducing the total volume of transmitted data. But in a high-bandwidth network, a setting of 1 might improve performance, if the CPU time required to process variable-length packets were greater than the time required to send the entire character stream. In cross-server distributed operations, this setting has no effect, and padding characters are dropped from VARCHAR or NVARCHAR values that are passed between database servers.

IFX_UNLOAD_EILSEQ_MODE environment variable

Use the IFX_UNLOAD_EILSEQ_MODE environment variable to help migrate databases from Informix Version 10 to Version 11.50 or 11.70, where character data might be encoded with a codeset that is different than the codeset used to create the Version 10 database.

In earlier versions of Informix, it was possible to load character data into a database that did not match the locale and codeset of the database. For example you could load Chinese data into a database created with the DB_LOCALE=en_US.8859-1 codeset. In newer versions of Informix, to insert Chinese data you would need a database created with the Chinese (DB_LOCALE=zh_tw.big5 locale and codeset.

Important: For databases created with Version 10 and Client SDK 2.4, when you attempt to unload the invalid character data an error occurs unless you have set this environment variable. The IFX_UNLOAD_EILSEQ_MODE environment variable enables DB-Access, dbexport, and High Performance Loader (HPL) to unload character and bypass the GLS validation that normally occurs when you unload data by using the Version 11.50 and 11.70 tools.

To use this environment variable, set it to any non-empty string.

```
►►──setenv──IFX_UNLOAD_EILSEQ_MODE──value────────────────────────────►◄
```

value Any alpha or numeric value. For example: yes, true, or 1.

This environment variable takes effect when character data is being fetched or retrieved from the database.

```
setenv IFX_UNLOAD_EILSEQ_MODE 1
setenv IFX_UNLOAD_EILSEQ_MODE yes
setenv IFX_UNLOAD_EILSEQ_MODE on
```

This environment variable is similar to setting the EILSEQ_COMPAT_MODE configuration parameter in the ONCONFIG file. The configuration parameter affects character data that is inserted into the database, whereas the IFX_UNLOAD_EILSEQ_MODE environment variable affects character data that is unloaded from the database.

IFX_UPDDESC environment variable

You must set the **IFX_UPDDESC** environment variable at execution time before you can do a DESCRIBE of an UPDATE statement.

```
►►──setenv──IFX_UPDDESC──value───────────────────────────────────────►◄
```

value is any non-NULL value.

A NULL value (here meaning that **IFX_UPDDESC** is not set) disables the describe-for-update feature. Any non-NULL value enables the feature.

IFX_XASTDCOMPLIANCE_XAEND environment variable

In earlier releases of IBM Informix, an internal rollback of a global transaction freed the transaction. In releases later than Version 9.40, however, the default behavior after an internal rollback is not to free the global transaction until an explicit rollback, as required by the X/Open XA standard. By setting the DISABLE_B162428_XA_FIX configuration parameter to 1, you can restore the legacy behavior as the default for all sessions.

The **IFX_XASTDCOMPLIANCE_XAEND** environment variable can override the configuration parameter for the current session, using the following syntax. Valid **IFX_XASTDCOMPLIANCE_XAEND** values are 1 and 0.

```
►►──setenv──IFX_XASTDCOMPLIANCE_XAEND──┬─1─┬──────────────────────────────►◄
                                       └─0─┘
```

0 Frees global transactions only after an explicit rollback

1 Frees global transactions after any rollback

This environment variable can be particularly useful when the server instance is disabled for new behavior by the DISABLE_B162428_XA_FIX configuration parameter, but one client requires the new behavior. Setting this environment variable to zero supports the new behavior in the current session.

IFX_XFER_SHMBASE environment variable

An alternative base address for a utility to attach the server shared memory segments.

```
►►──setenv──IFX_XFER_SHMBASE────address───────────────────────────────────►◄
```

address
Valid address in hexadecimal

After the database server allocates shared memory, the database server might allocate multiple contiguous OS shared memory segments. The client utility that connects to shared memory must attach all those OS segments contiguously also. The utility might have some other shared objects (for example, the xbsa library in onbar) loaded at the address where the server has shared memory segment attached. To workaround this situation, you can specify a different base address in the environment variable IFX_XFER_SHMBASE for the utility to attach the shared memory segments. The onstat, onmode, and oncheck utilities must attach to exact same shared memory base as oninit. Setting IFX_XFER_SHMBASE is not an option for these utilities.

IMCADMIN environment variable

The **IMCADMIN** environment variable supports the **imcadmin** administrative tool by specifying the name of a database server through which **imcadmin** can connect to MaxConnect. For **imcadmin** to operate correctly, you must set IMCADMIN before you use any IBM Informix products.

```
►►──setenv──IMCADMIN──dbservername──────────────────────────────────────►◄
```

dbservername
is the name of a database server.

Here *dbservername* must be listed in the **sqlhosts** file on the computer where the MaxConnect runs. MaxConnect uses this setting to obtain the following connectivity information from the **sqlhosts** file:

- Where the administrative listener port must be established
- The network protocol that the specified database server uses
- The host name of the system where the specified database server is located

You cannot use the **imcadmin** tool unless **IMCADMIN** is set to a valid database server name.

For more information about using **IMCADMIN**, see *IBM Informix MaxConnect User's Guide*.

IMCCONFIG environment variable

The **IMCCONFIG** environment variable specifies a nondefault filename, and optionally a pathname, for the MaxConnect configuration file. On UNIX systems that support the C shell, this variable can be set by the following command.

```
►►──setenv──IMCCONFIG──pathname─────────────────────────────────────────►◄
```

pathname
is a full pathname or a simple filename.

When the setting is a filename that is not qualified by a full pathname, MaxConnect searches for the specified file in the **$INFORMIXDIR/etc/** directory. Thus, if you set **IMCCONFIG** to **IMCconfig.imc2**, MaxConnect searches for **$INFORMIXDIR/etc/IMCconfig.imc2** as its configuration file.

If the **IMCCONFIG** environment variable is not set, MaxConnect searches by default for **$INFORMIXDIR/etc/IMCconfig** as its configuration file.

IMCSERVER environment variable

The **IMCSERVER** environment variable specifies the name of a database server entry in the **sqlhosts** file that contains information about connectivity.

The database server can be either local or remote. On UNIX systems that support the C shell, the **IMCSERVER** environment variable can be set by the command.

```
►►──setenv──IMCSERVER──dbservername─────────────────────────────────────►◄
```

dbservername
is the valid name of a database server.

Here *dbservername* must be the name of a database server in the **sqlhosts** file. For more information about **sqlhosts** settings with MaxConnect, see your *IBM Informix Administrator's Guide*. You cannot use MaxConnect unless **IMCSERVER** is set to a valid database server name.

INFORMIXC environment variable (UNIX)

The **INFORMIXC** environment variable specifies the filename or pathname of the C compiler to be used to compile files that IBM Informix ESQL/C generates. The setting takes effect only during the C compilation stage.

If **INFORMIXC** is not set, the default compiler on most systems is **cc**.

Tip: On Windows, you pass either `-mcc` or `-bcc` options to the *esql* preprocessor to use either the Microsoft or Borland C compilers.

```
►►──setenv──INFORMIXC──┬─compiler─┬──────────────────────────────────►◄
                       └─pathname─┘
```

compiler
: The file name of the C compiler.

pathname
: The full path name of the C compiler.

For example, to specify the GNU C compiler, enter the following command:

```
setenv INFORMIXC gcc
```

Important: If you use **gcc**, be aware that the database server assumes that strings are writable, so you must compile by using the `-fwritable-strings` option. Failure to do so can produce unpredictable results, possibly including core dumps.

INFORMIXCMNAME environment variable

If the Connection Manager raises an event alarm, the **INFORMIXCMNAME** environment variable is used to store the name of the Connection Manager instance that raised the alarm. The environment variable is set automatically by the Connection Manager.

The **INFORMIXCMNAME** environment variable corresponds to the NAME parameter in the Connection Manager configuration file. The environment variable is used by the **CMALARMPROGRAM** program to determine the Connection Manager instance responsible for the event alarm. You can also use the environment variable in your own Connection Manager event alarm handler.

The environment variable is set automatically by the Connection Manager and should not be modified.

Related reference:

"INFORMIXCMCONUNITNAME environment variable"

Related information:

The oncmsm utility

Connection Manager event alarm IDs

INFORMIXCMCONUNITNAME environment variable

If the Connection Manager raises an event alarm, the **INFORMIXCMCONUNITNAME** environment variable is used to store the name of the Connection Manager connection unit that raised the alarm. The environment variable is set automatically by the Connection Manager.

The **INFORMIXCMCONUNITNAME** environment variable corresponds to the connection unit name parameter in the Connection Manager configuration file. The environment variable is used by the **CMALARMPROGRAM** program to determine the Connection Manager instance responsible for the event alarm. You can also use the environment variable in your own Connection Manager event alarm handler.

The environment variable is set automatically by the Connection Manager and should not be modified.

Related reference:

"INFORMIXCMNAME environment variable" on page 3-50

Related information:

The oncmsm utility

Connection Manager event alarm IDs

INFORMIXCONCSMCFG environment variable

Use the **INFORMIXCONCSMCFG** environment variable to specify the location of the **concsm.cfg** file that describes communications support modules.

```
►►──setenv──INFORMIXCONCSMCFG──pathname──────────────────────────────►◄
```

pathname
specifies the full pathname of the **concsm.cfg** file.

The following command specifies that the **concsm.cfg** file is in **/usr/myfiles**:

```
setenv INFORMIXCONCSMCFG /usr/myfiles
```

You can also specify a different name for the file. The following example specifies a filename of **csmconfig** in the same directory:

```
setenv INFORMIXCONCSMCFG /usr/myfiles/csmconfig
```

The default location of the **concsm.cfg** file is in **$INFORMIXDIR/etc**. For more information about communications support modules and the contents of the **concsm.cfg** file, see the *IBM Informix Administrator's Reference*.

INFORMIXCONRETRY environment variable

The **INFORMIXCONRETRY** environment variable sets a limit on the maximum number of connection attempts that can be made to each database server by the client after the initial connection attempt fails. These attempts are made within the time limit that the **INFORMIXCONTIME** setting specifies.

```
►►──setenv──INFORMIXCONRETRY──count──────────────────────────────────►◄
```

count The number of additional attempts to connect to each database server after the initial connection attempt fails.

For example, the following command sets **INFORMIXCONRETRY** to specify three connection attempts after the initial attempt:

```
setenv INFORMIXCONRETRY 3
```

The default value for **INFORMIXCONRETRY** is one attempt after the initial connection attempt.

Order of precedence among INFORMIXCONRETRY settings

When you specify a setting for the **INFORMIXCONRETRY** client environment variable, it overrides any **INFORMIXCONRETRY** configuration parameter setting in the `onconfig` file.

If the SET ENVIRONMENT statement specifies a setting for the INFORMIXCONRETRY session environment option, however, the SQL statement setting overrides the **INFORMIXCONRETRY** client environment variable setting for subsequent connection attempts during the current session. The SET ENVIRONMENT INFORMIXCONRETRY setting has no effect on other sessions.

In summary, this is the ascending order (lowest to highest) of the methods for setting a limit on attempts for a connection to a database server:
- **INFORMIXCONRETRY** configuration parameter
- **INFORMIXCONRETRY** client environment variable
- SET ENVIRONMENT INFORMIXCONRETRY statement of SQL

The **INFORMIXCONTIME** setting takes precedence over the **INFORMIXCONRETRY** setting. Connection attempts can end after the **INFORMIXCONTIME** value is exceeded, but before the **INFORMIXCONRETRY** value is reached. For more information about restricting the time available to establish a connection to a database server, see "INFORMIXCONTIME environment variable"

Related reference:

INFORMIXCONRETRY configuration parameter

Related information:

INFORMIXCONRETRY session environment option

INFORMIXCONTIME environment variable

The **INFORMIXCONTIME** environment variable specifies the number of seconds the CONNECT statement attempts to establish a connection to a database server before returning an error. If you set no value, the default of 60 seconds can typically support a few hundred concurrent client connections. However, some systems might encounter few connection errors with a value as low as 15. The total distance between nodes, hardware speed, the volume of traffic, and the concurrency level of the network can all affect what value you should set to optimize **INFORMIXCONTIME**.

The **INFORMIXCONTIME** and **INFORMIXCONRETRY** environment variables let you configure your client-side connection capability to retry the connection instead of returning a **-908** error.

```
►►──setenv──INFORMIXCONTIME──seconds──────────────────────────────────────►◄
```

seconds
: Represents the minimum number of seconds spent in attempts to establish a connection to a database server.

For example, enter this command to set **INFORMIXCONTIME** to `60` seconds:

```
setenv INFORMIXCONTIME 60
```

If **INFORMIXCONTIME** is set to `60` and **INFORMIXCONRETRY** is set to `3`, attempts to connect to the database server (after the initial attempt at `0` seconds) are made at

20, 40, and 60 seconds, if necessary, before aborting. This 20-second interval is the result of **INFORMIXCONTIME** divided by **INFORMIXCONRETRY**. If you set the **INFORMIXCONTIME** value to zero, the database server automatically uses the default value of 60 seconds.

If the CONNECT statement must search **DBPATH**, the **INFORMIXCONRETRY** setting specifies the number of additional connection attempts that can be made for each database server entry in **DBPATH**.

- All appropriate servers in the **DBPATH** setting are accessed at least once, even if the **INFORMIXCONTIME** value is exceeded. Thus, the CONNECT statement might take longer than the **INFORMIXCONTIME** time limit to return an error that indicates connection failure or that the database was not found.
- The **INFORMIXCONTIME** value is divided among the number of database server entries that are specified in **DBPATH**. Thus, if **DBPATH** contains numerous servers, increase the **INFORMIXCONTIME** value accordingly. For example, if **DBPATH** contains three entries, to spend at least 30 seconds attempting each connection, set **INFORMIXCONTIME** to 90.

The **INFORMIXCONTIME** and **INFORMIXCONRETRY** environment variables can be modified with the **onutil** SET command, as in the following example:

```
% onutil
1> SET INFORMIXCONTIME 120;
Dynamic Configuration completed successfully
2> SET INFORMIXCONRETRY 10;
Dynamic Configuration completed successfully
```

Order of precedence among INFORMIXCONTIME settings

When you specify a setting for the **INFORMIXCONTIME** client environment variable, it overrides the **INFORMIXCONTIME** configuration parameter settings in the onconfig file for the current session.

If the SET ENVIRONMENT statement specifies a setting for the INFORMIXCONRETRY session environment option, however, the SQL statement setting overrides the **INFORMIXCONRETRY** client environment variable setting for subsequent connection attempts during the current session. The SET ENVIRONMENT INFORMIXCONRETRY setting has no effect on other sessions.

In summary, this is the ascending order (lowest to highest) of the methods for setting an upper limit on the amount of time that a CONNECT statement can spend attempting to connect to a database server:

- **INFORMIXCONTIME** configuration parameter
- **INFORMIXCONTIME** client environment variable
- SET ENVIRONMENT INFORMIXCONTIME statement of SQL.

INFORMIXCONTIME takes precedence over the **INFORMIXCONRETRY** setting. Connection attempts can end after the **INFORMIXCONTIME** value is exceeded, but before the **INFORMIXCONRETRY** value is reached.

Related information:

INFORMIXCONTIME session environment option

INFORMIXCONTIME configuration parameter

INFORMIXCPPMAP environment variable

Set the **INFORMIXCPPMAP** environment variable to specify the fully qualified pathname of the map file for C++ programs. Information in the map file includes the database server type, the name of the shared library that supports the database object or value object type, the library entry point for the object, and the C++ library for which an object was built.

```
►►—setenv—INFORMIXCPPMAP—pathname——————————————————————————————————►◄
```

pathname
The directory path where the C++ map file is stored.

The map file is a text file that can have any filename. You can specify several map files, separated by colons (**:**) on UNIX or semicolons (**;**) on Windows.

On UNIX, the default map file is `$INFORMIXDIR/etc/c++map`. On Windows, the default map file is `%INFORMIXDIR%\etc\c++map`.

INFORMIXDIR environment variable

The **INFORMIXDIR** environment variable specifies the directory that contains the subdirectories in which your product files are installed. You must always set **INFORMIXDIR**. Verify that **INFORMIXDIR** is set to the full pathname of the directory in which you installed your database server. If you have multiple versions of a database server, set **INFORMIXDIR** to the appropriate directory name for the version that you want to access. For information about when to set **INFORMIXDIR**, see your *IBM Informix Installation Guide*.

```
►►—setenv—INFORMIXDIR\—pathname————————————————————————————————————►◄
```

pathname
is the directory path where the product files are installed.

To set **INFORMIXDIR** to **usr/informix/**, for example, as the installation directory, enter the following command:

```
setenv INFORMIXDIR /usr/informix
```

INFORMIXSERVER environment variable

The **INFORMIXSERVER** environment variable specifies the default database server to which an explicit or implicit connection is made by an SQL API client, the DB-Access utility, or other IBM Informix products.

This environment variable must be set before you can use IBM Informix client products. It has the following syntax.

```
►►—setenv—INFORMIXSERVER—dbservername——————————————————————————————►◄
```

dbservername
is the name of the default database server.

The value of **INFORMIXSERVER** can be a local or remote server, but must correspond to a valid *dbservername* entry in the **$INFORMIXDIR/etc/sqlhosts** file on the computer running the application. The *dbservername* must begin with a lower-case letter and cannot exceed 128 bytes. It can include any printable characters except uppercase characters, field delimiters (blank space or tab), the newline character, and the hyphen (or minus) symbol.

For example, this command specifies the **coral** database server as the default:

```
setenv INFORMIXSERVER coral
```

INFORMIXSERVER specifies the database server to which an application connects if the CONNECT DEFAULT statement is executed. It also defines the database server to which an initial implicit connection is established if the first statement in an application is not a CONNECT statement.

Important: You must set **INFORMIXSERVER** even if the application or DB-Access does not use implicit or explicit default connections.

INFORMIXSHMBASE environment variable (UNIX)

The **INFORMIXSHMBASE** environment variable affects only client applications connected to IBM Informix databases that use the interprocess communications (IPC) shared-memory (**ipcshm**) protocol.

Important: Resetting **INFORMIXSHMBASE** requires a thorough understanding of how the application uses memory. Normally you do not reset **INFORMIXSHMBASE**.

INFORMIXSHMBASE specifies where shared-memory communication segments are attached to the client process so that client applications can avoid collisions with other memory segments that it uses. If you do not set **INFORMIXSHMBASE**, the memory address of the communication segments defaults to an implementation-specific value such as 0x800000.

```
►►──setenv──INFORMIXSHMBASE──value──────────────────────────────────────►◄
```

value is an integer (in KB) used to calculate the memory address.

The database server calculates the memory address where segments are attached by multiplying the value of **INFORMIXSHMBASE** by 1,024. For example, on a system that uses the C shell, you can set the memory address to the value 0x800000 by entering the following command:

```
setenv INFORMIXSHMBASE 8192
```

For more information, see your *IBM Informix Administrator's Guide* and the *IBM Informix Administrator's Reference*.

INFORMIXSQLHOSTS environment variable

The **INFORMIXSQLHOSTS** environment variable specifies where the SQL client or the database server can find connectivity information.

```
►►──setenv──INFORMIXSQLHOSTS──pathname──────────────────────────────────►◄
```

pathname
The full path name of the connectivity information file.

UNIX: Default =$INFORMIXDIR/etc/sqlhosts

Windows server: Default = `%INFORMIXDIR%\etc\sqlhosts.` `%INFORMIXSERVER%`

For example, the following command overrides the default location and specifies that the `mysqlhosts` file is in the `/work/envt` directory:

```
setenv INFORMIXSQLHOSTS /work/envt/mysqlhosts
```

Windows client: The INFORMIXSQLHOSTS environment variable points to the computer whose registry contains the SQLHOSTS subkey. For example, the following command instructs the Windows client to look for connectivity information in the registry of a computer named **arizona**:

```
set INFORMIXSQLHOSTS = \\arizona
```

INFORMIXSTACKSIZE environment variable

The **INFORMIXSTACKSIZE** environment variable specifies the stack size (in KB) that is applied to all client processes. Any value that you set for INFORMIXSTACKSIZE in the client environment is ignored by the database server.

```
►►──setenv──INFORMIXSTACKSIZE──size──────────────────────────────────►◄
```

size is an integer, setting the stack size (in KB) for SQL client threads.

For example, to decrease the **INFORMIXSTACKSIZE** to 20 KB, enter the following command:

```
setenv INFORMIXSTACKSIZE 20
```

If **INFORMIXSTACKSIZE** is not set, the stack size is taken from the database server configuration parameter STACKSIZE or else defaults to a platform-specific value. The default stack size value for the primary thread of an SQL client is 32 KB for nonrecursive database activity.

Warning: For instructions on setting this value, see the *IBM Informix Administrator's Reference*. If you incorrectly set the value of **INFORMIXSTACKSIZE**, it can cause the database server to fail.

INFORMIXTERM environment variable (UNIX)

The **`INFORMIXTERM`** environment variable specifies whether DB-Access should use the information in the `terminfo` directory or the `termcap` file.

On character-based systems, the `terminfo` directory and `termcap` file determine terminal-dependent keyboard and screen capabilities, such as the operation of function keys, color and intensity attributes in screen displays, and the definition of window borders and graphic characters.

```
►►──setenv──INFORMIXTERM──┬─terminfo─┬───────────────────────────────►◄
                          └─termcap──┘
```

If **`INFORMIXTERM`** is not set, the default setting is `terminfo`.

The `terminfo` directory contains a file for each terminal name that has been defined. The `terminfo` setting for **`INFORMIXTERM`** is supported only on computers that provide full support for the UNIX System V `terminfo` library. For details, see the machine notes file for your product.

When DB-Access is installed on your system, a `termcap` file is placed in the **etc** subdirectory of `$INFORMIXDIR`. This file is a superset of an operating-system **termcap** file. You can use the **termcap** file that the database server supplies, the system `termcap` file, or a `termcap` file that you create. You must set the **`TERMCAP`** environment variable if you do not use the default `termcap` file. For information about setting the **`TERMCAP`** environment variable, see "TERMCAP environment variable (UNIX)" on page 3-72.

INF_ROLE_SEP environment variable

The **`INF_ROLE_SEP`** environment variable configures the security feature of role separation when the database server is installed or reinstalled on UNIX systems. Role separation enforces separating administrative tasks by people who run and audit the database server. After the installation is complete, **`INF_ROLE_SEP`** has no effect. If **`INF_ROLE_SEP`** is not set, then user **informix** (the default) can perform all administrative tasks.

```
►►──setenv──INF_ROLE_SEP──n──────────────────────────────────────────►◄
```

n is any positive integer.

On Windows, the install process asks whether you want to enable role separation regardless of the setting of **`INF_ROLE_SEP`**. To enable role separation for database servers on Windows, select the role-separation option during installation.

If **`INF_ROLE_SEP`** is set when IBM Informix is installed on a UNIX platform, role separation is implemented and a separate group is specified to serve each of the following responsibilities:
- The Database Server Administrator (DBSA)
- The Audit Analysis Officer (AAO)
- The standard user

On UNIX, you can establish role separation by changing the group that owns the `aaodir`, `dbsadir`, or `etc` directories at any time after the installation is complete. You can disable role separation by resetting the group that owns these directories to **informix**. You can have role separation enabled, for example, for the Audit Analysis Officer (AAO) without having role separation enabled for the Database Server Administrator (DBSA).

For more information about the security feature of role separation, see the *IBM Informix Security Guide*. To learn how to configure role separation when you install your database server, see your *IBM Informix Installation Guide*.

INTERACTIVE_DESKTOP_OFF environment variable (Windows)

This environment variable lets you prevent interaction with the Windows desktop when an SPL routine executes a SYSTEM command.

```
►►──setenv──INTERACTIVE_DESKTOP_OFF──┬─1─┬───────────────────────────►◄
                                     └─0─┘
```

If **INTERACTIVE_DESKTOP_OFF** is 1 and an SPL routine attempts to interact with the desktop (for example, with the `notepad.exe` or `cmd.exe` program), the routine fails unless the user is a member of the **Administrators** group.

The valid settings (`1` or `0`) have the following effects:

1	Prevents the database server from acquiring desktop resources for the user executing the stored procedure
0	SYSTEM commands in a stored procedure can interact with the desktop. This is the default value.

Setting **INTERACTIVE_DESKTOP_OFF** to 1 allows an SPL routine that does not interact with the desktop to execute more quickly. This setting also allows the database server to simultaneously call a greater number of SYSTEM commands because the command no longer depends on a limited operating- system resource (Desktop and WindowStation handles).

JAR_TEMP_PATH environment variable

Set the **JAR_TEMP_PATH** variable to specify a non-default local file system location where jar management procedures such as **install_jar( )** and **replace_jar( )** can store temporary **.jar** files of the Java virtual machine.

```
►►──setenv──JAR_TEMP_PATH──pathname──────────────────────────────────────►◄
```

pathname
specifies a local directory for temporary **.jar** files.

This directory must have read and write permissions for the user who starts the database server. If the **JAR_TEMP_PATH** environment variable is not set, temporary copies of **.jar** files are stored in the **/tmp** directory of the local file system for the database server.

JAVA_COMPILER environment variable

You can set the **`JAVA_COMPILER`** environment variable in the Java virtual machine environment to disable JIT compilation.

```
►►──setenv──JAVA_COMPILER──┬─none─┬──────────────────────────────────────►◄
                           └─NONE─┘
```

The `NONE` and `none` settings are equivalent. On UNIX systems that support the C shell and on which **`JAVA_COMPILER`** has been set to `NONE` or `none`, you can enable the JIT compiler for the JVM environment by the following command:

```
unset JAVA_COMPILER
```

JVM_MAX_HEAP_SIZE environment variable

The **JVM_MAX_HEAP_SIZE** environment variable can set a non-default upper limit on the size of the heap for the Java virtual machine.

```
►►──setenv──JVM_MAX_HEAP_SIZE──size──────────────────────────────────────►◄
```

size is a positive integer that specifies the maximum size (in megabytes).

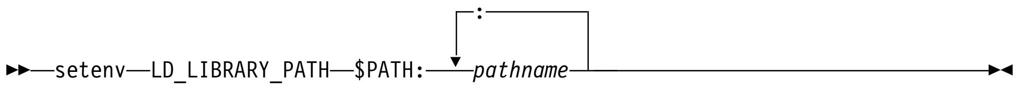
For example, the following command sets the maximum heap size at 12 MB:

```
set JVM_MAX_HEAP_SIZE 12
```

If you do not set **JVM_MAX_HEAP_SIZE**, 16 MB is the default maximum size.

LD_LIBRARY_PATH environment variable (UNIX)

The **LD_LIBRARY_PATH** environment variable tells the shell on Solaris systems which directories to search for client or shared IBM Informix general libraries. You must specify the directory that contains your client libraries before you can use the product.



pathname
Specifies the search path for the library.

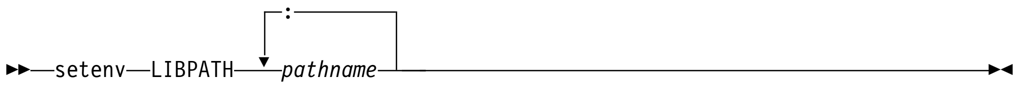
For INTERSOLV DataDirect ODBC Driver on AIX, set **LIBPATH**. For INTERSOLV DataDirect ODBC Driver on HP-UX, set **SHLIB_PATH**.

The following example sets the **LD_LIBRARY_PATH** environment variable to the directory:

```
setenv LD_LIBRARY_PATH
${INFORMIXDIR}/lib:${INFORMIXDIR}/lib/esql:$LD_LIBRARY_PATH
```

LIBPATH environment variable (UNIX)

The **LIBPATH** environment variable tells the shell on AIX systems which directories to search for dynamic-link libraries for the INTERSOLV DataDirect ODBC Driver. You must specify the full path name for the directory where you installed the product.



pathname
Specifies the search path for the libraries.

On Solaris, set **LD_LIBRARY_PATH**. On HP-UX, set **SHLIB_PATH**.

NODEFDAC environment variable

Enabling NODEFDAC applies the ANSI-compliant restrictions on default access privileges for the PUBLIC group when tables or Owner-mode user-defined routines are created in databases that are not ANSI-compliant.

In a database that is not ANSI-compliant, when the **NODEFDAC** environment variable enabled by setting it to yes,

- the database server withholds default table access privileges from PUBLIC when a new table is created,
- and also withholds the default Execute privilege from PUBLIC when an owner-privileged UDR is created.

```
►►—setenv—NODEFDAC——yes——————————————————————————————————————►◄
```

yes prevents default table privileges (Select, Insert, Update, and Delete) from being granted to PUBLIC on new tables in a database that is not ANSI-compliant. This setting also prevents the Execute privilege from being granted to PUBLIC by default when a new user-defined routine is created in Owner mode.

The *yes* setting is case sensitive, and is also sensitive to leading and trailing blank spaces. Including uppercase letters or blank spaces in the setting is equivalent to leaving **NODEFDAC** unset. When **NODEFDAC** is not set, or if it is set to any value besides *yes*, default privileges on tables and Owner-mode UDRs are granted to PUBLIC by default when the table or UDR is created in a database that is not ANSI-compliant. The setting `YES`, for example, disables **NODEFDAC**.

Enabling **NODEFDAC** has no effect in an ANSI-compliant databases.

Important: Enabling **NODEFDAC** withholds default table or routine privileges from PUBLIC when the object is created, but the **NODEFDAC** setting cannot prevent the PUBLIC group from being granted the same privileges by a user who holds the necessary access privileges on the new table or on the new UDR.

ONCONFIG environment variable

The **`ONCONFIG`** environment variable specifies the name of the active file, called the `onconfig` file, which holds the configuration parameters for the database server.

This file is read as input during the initialization procedure. After you prepare your `onconfig` configuration file, set the **`ONCONFIG`** environment variable to the name of this file.

```
►►—setenv—ONCONFIG—filename——————————————————————————————————►◄
```

filename
is the name of your `onconfig` file in the `%INFORMIXDIR%\etc\%ONCONFIG%` or `$INFORMIXDIR/etc/$ONCONFIG` directory

This file contains the configuration parameters for your database.

To prepare the `onconfig` file, make a copy of the `onconfig.std` file and modify the copy. Name the `onconfig` file so that it can easily be related to a specific database server. If you have multiple instances of a database server, each instance *must* have its own uniquely named `onconfig` file.

If the **`ONCONFIG`** environment variable is not set, the database server reads the configuration values from the `onconfig` file during initialization.

ONINIT_STDOUT environment variable (Windows)

The **`ONINIT_STDOUT`** environment variable specifies a path and file name in which output from the **oninit** command is stored.

While it is not generally necessary to view output from the **oninit** command, it might be necessary in certain situations, such as when using the `-v` (verbose) option or when you want to see output from an unhandled exception in a process launched within a virtual processor. When the value of **ONINIT_STDOUT** is set to the name of a file, output from the **oninit** command is written to the file.

```
►►──set──ONINIT_STDOUT\──path\filename──────────────────────────────────►◄
```

You can set the **ONINIT_STDOUT** environment variable as a system variable in **Control Panel** > **System** > **Advanced** > **Environment Variables**. If the IBM Informix service is configured to log on as user **informix**, start the service using the **starts** command after setting the environment variable. Note, however, that because environment variables are read from the system when the service is started, if the service is set to log on as the local system user, you must restart your computer for the environment variable to take effect. Because the local system user is effectively logged on at all times, environment variables are refreshed only when the operating system is restarted.

For example, if the environment variable set to `C:\temp\oninit_out.txt`, you can start the server with the verbose option with the following command:

```
starts %INFORMIXSERVER% -v
```

The **oninit** messages are saved to the `C:\temp\oninit_out.txt` file.

Important: Only a single instance of the database can run on a Windows machine if the **ONINIT_STDOUT** environment variable is set.

OPTCOMPIND environment variable

You can set the **OPTCOMPIND** environment variable so that the optimizer can select the appropriate join method.

```
                           ┌─2─┐
►►──setenv──OPTCOMPIND─────┼─1─┼────────────────────────────────────────►◄
                           └─0─┘
```

0 A nested-loop join is preferred, where possible, over a sort-merge join or a hash join.

1 When the isolation level is *not* Repeatable Read, the optimizer behaves as in setting `2`; otherwise, the optimizer behaves as in setting `0`.

2 Nested-loop joins are not necessarily preferred. The optimizer bases its decision purely on costs, regardless of transaction isolation mode.

When **OPTCOMPIND** is not set, the database server uses the OPTCOMPIND value from the ONCONFIG configuration file. When neither the environment variable nor the configuration parameter is set, the default value is `2`.

On IBM Informix, the SET ENVIRONMENT OPTCOMPIND statement can set or reset **OPTCOMPIND** dynamically at runtime. This overrides the current **OPTCOMPIND** value (or the ONCONFIG configuration parameter OPTCOMPIND) for the current user session only. For more information about the SET ENVIRONMENT OPCOMPIND statement of SQL see the *IBM Informix Guide to SQL: Syntax*.

For more information about the ONCONFIG configuration parameter OPTCOMPIND, see the *IBM Informix Administrator's Reference*. For more information about the different join methods that the optimizer uses, see your *IBM Informix Performance Guide*.

OPTMSG environment variable

Set the **OPTMSG** environment variable at runtime before you start the IBM Informix ESQL/C application to enable (or disable) optimized message transfers (message chaining) for all SQL statements in the application.

```
                  ┌─0─┐
►►──setenv──OPTMSG──┴─1─┴──────────────────────────────────────────►◄
```

0 disables optimized message transfers.

1 enables optimized message transfers and implements the feature for any subsequent connection.

The default value is `0` (zero), which explicitly disables message chaining. You might want, for example, to disable optimized message transfers for statements that require immediate replies, for debugging, or to ensure that the database server processes all messages before the application terminates.

When you set **OPTMSG** within an application, you can activate or deactivate optimized message transfers for each connection or within each thread. To enable optimized message transfers, you must set **OPTMSG** before you establish a connection.

For more information about setting **OPTMSG** and defining related global variables, see the *IBM Informix ESQL/C Programmer's Manual*.

OPTOFC environment variable

Use the **OPTOFC** environment variable to enable optimize-OPEN-FETCH-CLOSE functionality in IBM Informix ESQL/C applications or other APIs (such as JDBC, ODBC, OLE DB, LIBDMI, and Lib C++) that use DECLARE and OPEN statements to establish a cursor.

```
                  ┌─0─┐
►►──setenv──OPTOFC──┴─1─┴──────────────────────────────────────────►◄
```

0 disables **OPTOFC** for all threads of the application.

1 enables **OPTOFC** for every cursor in every thread of the application.

The default value is `0` (zero).

You can set the **OPTOFC** environment variable on the client or server. If this environment variable is set on the server, then any application that does not explicitly set this environment variable uses the value that is set on the server.

The **OPTOFC** environment variable reduces the number of message requests between the application and the database server.

If you set **OPTOFC** from the shell, you must set it before you start the Informix ESQL/C application. For more information about enabling **OPTOFC** and related features, see the *IBM Informix ESQL/C Programmer's Manual*.

OPT_GOAL environment variable (UNIX)

Set the **`OPT_GOAL`** environment variable in the user environment, before you start an application, to specify the query performance goal for the optimizer.

```
                        ┌─-1─┐
►►──setenv──OPT_GOAL────┴─0──┴──────────────────────────────────────────►◄
```

0 Specifies user-response-time optimization.

-1 Specifies total-query-time optimization.

The default behavior is for the optimizer to use query plans that optimize the total query time.

You can also specify the optimization goal for individual queries with optimizer directives or for a session with the SET OPTIMIZATION statement.

Both methods take precedence over the **`OPT_GOAL`** environment variable setting. You can also set the OPT_GOAL configuration parameter for the IBM Informix system; this method has the lowest level of precedence.

For more information about optimizing queries for your database server, see your *IBM Informix Performance Guide*. For information about the SET OPTIMIZATION statement, see the *IBM Informix Guide to SQL: Syntax*.

PATH environment variable

The UNIX **`PATH`** environment variable tells the shell which directories to search for executable programs. You must add the directory containing your IBM Informix product to your **`PATH`** setting before you can use the product.

```
                            ┌─:────────┐
                            ▼          │
►►──setenv──PATH──$PATH:────pathname───┴────────────────────────────────►◄
```

pathname
Specifies the search path for the executable files.

Include a colon (:) separator between the path names on UNIX systems. (Use the semicolon (;) separator between path names on Windows systems.)

You can specify the search path in various ways. The **`PATH`** environment variable tells the operating system where to search for executable programs. You must include the directory that contains your IBM Informix product in your **path** setting before you can use the product. This directory should be located before `$INFORMIXDIR/bin`, which you must also include.

For additional information about how to modify your path, see "Modifying an environment-variable setting" on page 3-4.

PDQPRIORITY environment variable

The **PDQPRIORITY** environment variable determines the degree of parallelism that the database server uses and affects how the database server allocates resources, including memory, processors, and disk reads.

```
►►──setenv──PDQPRIORITY──┬─HIGH──────┬──────────────────────────────►◄
                         ├─LOW───────┤
                         ├─OFF───────┤
                         └─resources─┘
```

resources
: Is an integer in the range `0` to `100`. The value `1` is the same as LOW, and `100` is the same as HIGH. Values lower than `0` are set to `0` (OFF), and values greater than `100` are set to `100` (HIGH).

  Value `0` is the same as OFF (for IBM Informix only).

Here the HIGH, LOW, and OFF keywords have the following effects:

HIGH When the database server allocates resources among all users, it gives as many resources as possible to the query.

LOW Data values are fetched from fragmented tables in parallel.

OFF PDQ processing is turned off (for IBM Informix only).

Usually, the more resources a database server uses, the better its performance for a given query. If the server uses too many resources, however, contention for the resources can take resources away from other queries, resulting in degraded performance. For more information about performance considerations for **PDQPRIORITY**, see the *IBM Informix Performance Guide*.

An application can override the setting of this environment variable when it issues the SQL statement SET PDQPRIORITY, as the *IBM Informix Guide to SQL: Syntax* describes.

Using PDQPRIORITY with Informix

The *resources* value specifies the query priority level and the amount of resources that the database server uses to process the query.

When **`PDQPRIORITY`** is not set, the default value is `OFF`.

When **`PDQPRIORITY`** is set to HIGH, IBM Informix determines an appropriate value to use for **`PDQPRIORITY`** based on several criteria. These include the number of available processors, the fragmentation of tables queried, the complexity of the query, and additional factors.

PLCONFIG environment variable

The **`PLCONFIG`** environment variable specifies the name of the configuration file that the High Performance Loader (HPL) uses. This file must be located in the `$INFORMIXDIR/etc` directory. If the **`PLCONFIG`** environment variable is not set, then `$INFORMIXDIR/etc/plconfig` is the default configuration file.

```
►►──setenv──PLCONFIG──filename──────────────────────────────────────►◄
```

filename
Specifies the simple file name of the configuration file that the High-Performance Loader uses.

For example, to specify the `$INFORMIXDIR/etc/custom.cfg` file as the configuration file for the High-Performance Loader, enter the following command:

```
setenv PLCONFIG custom.cfg
```

For more information, see the *IBM Informix High-Performance Loader User's Guide*.

PLOAD_LO_PATH environment variable

The **PLOAD_LO_PATH** environment variable lets you specify the pathname for smart-large-object handles (which identify the location of smart large objects such as BLOB and CLOB data types).

```
►►──setenv──PLOAD_LO_PATH──pathname────────────────────────────────────►◄
```

pathname
specifies the directory for the smart-large-object handles.

If **PLOAD_LO_PATH** is not set, the default directory is **/tmp**.

For more information, see the *IBM Informix High-Performance Loader User's Guide*.

PLOAD_SHMBASE environment variable

The **PLOAD_SHMBASE** environment variable lets you specify the shared-memory address at which the High Performance Loader (HPL) **onpload** processes will attach. If **PLOAD_SHMBASE** is not set, the HPL determines which shared-memory address to use.

```
►►──setenv──PLOAD_SHMBASE──value───────────────────────────────────────►◄
```

value Used to calculate the shared-memory address.

If the **onpload** utility cannot attach, an error is issued, and you must specify a new value.

The **onpload** utility tries to determine at which address to attach, as follows in the following (descending) order:

1. Attach at the same address (SHMBASE) as the database server.
2. Attach beyond the database server segments.
3. Attach at the address specified in **PLOAD_SHMBASE**.

Tip: It is recommended that you let the HPL decide where to attach and that you set **PLOAD_SHMBASE** only if necessary to avoid shared-memory collisions between **onpload** and the database server.

For more information, see the *IBM Informix High-Performance Loader User's Guide*.

PSM_ACT_LOG environment variable

Use the **PSM_ACT_LOG** environment variable to specify the location of the IBM Informix Primary Storage Manager activity log for your environment, for example, for a single session.

```
►►──setenv──PSM_ACT_LOG──pathname──────────────────────────────────►◄
```

pathname
: The full path name for the location of the `$INFORMIXDIR/psm_act.log`. If you specify a file name only, the storage manager creates the activity log in the working directory in which you started the storage manager.

The **PSM_ACT_LOG** environment variable overrides the value of the PSM_ACT_LOG configuration parameter.

Related information:

PSM_ACT_LOG configuration parameter

PSM_CATALOG_PATH environment variable

Use the **PSM_CATALOG_PATH** environment variable to specify the location of the IBM Informix Primary Storage Manager catalog tables for your environment, for example, for a single session.

```
►►──setenv──PSM_CATALOG_PATH──pathname─────────────────────────────►◄
```

pathname
: The full path name for the location of the catalog table, which contain information about the pools, devices, and objects managed by the storage manager.

The **PSM_CATALOG_PATH** environment variable overrides the value of the PSM_CATALOG_PATH configuration parameter.

Related information:

PSM_CATALOG_PATH configuration parameter

PSM_DBS_POOL environment variable

Use the **PSM_DBS_POOL** environment variable to change the name of the pool in which the IBM Informix Primary Storage Manager places backup and restore dbspace data for your environment, for example, for a single session.

```
►►──setenv──PSM_DBS_POOL──pool_name────────────────────────────────►◄
```

pool_name
: The name of the storage manager pool.

The **PSM_DBS_POOL** environment variable overrides the value of the PSM_DBS_POOL configuration parameter.

Related information:

PSM_DBS_POOL configuration parameter

PSM_DEBUG environment variable

Use the **PSM_DEBUG** environment variable to specify the amount of debugging information that prints in the Informix Primary Storage Manager debug log for your environment, for example, for a single session.

```
►►──setenv──PSM_DEBUG──value──────────────────────────────────────────►◄
```

value 0 = No debugging messages.

1 = Prints only internal errors.

2 = Prints information about the entry and exit of functions and prints internal errors.

3 = Prints the information specified by 1-2 with additional details.

4 = Prints information about parallel operations and the information specified by 1-3.

5 = Prints information about internal states in the Informix Primary Storage Manager.

6 = Prints the information specified by 1-5 with additional details.

7 = Prints information specified by 1-6 with additional details.

8 = Prints information specified by 1-7 with additional details.

9 = Prints all debugging information.

The **PSM_DEBUG** environment variable overrides the value of the PSM_DEBUG configuration parameter.

Related information:

PSM_DEBUG configuration parameter

PSM_DEBUG_LOG environment variable

Use the **PSM_DEBUG_LOG** environment variable to specify the location of the IBM Informix Primary Storage Manager debug log for your environment, for example, for a single session.

```
►►──setenv──PSM_DEBUG_LOG──pathname───────────────────────────────────►◄
```

pathname
The full path name for the location of the `$INFORMIXDIR/psm_debug.log`. If you specify a file name only, the storage manager creates the debug log in the working directory in which you started the storage manager.

The **PSM_DEBUG_LOG** environment variable overrides the value of the PSM_DEBUG_LOG configuration parameter.

Related information:

PSM_DEBUG_LOG configuration parameter

PSM_LOG_POOL environment variable

Use the **PSM_LOG_POOL** environment variable to change the name of the pool in which the IBM Informix Primary Storage Manager places backup and restore log data for your environment, for example, for a single session.

```
►►──setenv──PSM_LOG_POOL──pool_name──────────────────────────────────►◄
```

pool_name
: The name of the storage manager log pool.

The **PSM_LOG_POOL** environment variable overrides the value of the PSM_LOG_POOL configuration parameter.

Related information:

PSM_LOG_POOL configuration parameter

PSORT_DBTEMP environment variable

The **PSORT_DBTEMP** environment variable specifies the location where the database server writes the temporary files that the **PSORT_NPROCS** environment variable uses to perform a sort.

```
                            ┌─:────────┐
                            ▼          │
►►──setenv──PSORT_DBTEMP────pathname───┴─────────────────────────────►◄
```

pathname
: The name of the UNIX directory used for intermediate writes during a sort.

To set the **PSORT_DBTEMP** environment variable to specify the directory (for example, /usr/leif/tempsort), enter the following command:

```
setenv PSORT_DBTEMP /usr/leif/tempsort
```

For maximum performance, specify directories that are located in file systems on different disks.

You might also want to consider setting the environment variable **DBSPACETEMP** to place temporary files used in sorting in dbspaces rather than operating-system files. See the discussion of the **DBSPACETEMP** environment variable in "DBSPACETEMP environment variable" on page 3-30.

The database server uses the directory that **PSORT_DBTEMP** specifies, even if the environment variable **PSORT_NPROCS** is not set. For additional information about the **PSORT_DBTEMP** environment variable, see your *IBM Informix Administrator's Guide* and your *IBM Informix Performance Guide*.

PSORT_NPROCS environment variable

The **PSORT_NPROCS** environment variable enables the database server to improve the performance of the parallel-process sorting package by allocating more threads for sorting.

Before the sorting package performs a parallel sort, make sure that the database server has enough memory for the sort.

```
►►──setenv──PSORT_NPROCS──threads────────────────────────────────────►◄
```

threads is an integer, specifying the maximum number of threads to be used to sort a query. This value cannot be greater than 10.

The following command sets **PSORT_NPROCS** to 4:

```
setenv PSORT_NPROCS 4
```

To disable parallel sorting, enter the following command:

```
unsetenv PSORT_NPROCS
```

It is recommended that you initially set **PSORT_NPROCS** to 2 when your computer has multiple CPUs. If subsequent CPU activity is lower than I/O activity, you can increase the value of **PSORT_NPROCS**.

Tip: If the **PDQPRIORITY** environment variable is not set, the database server allocates the minimum amount of memory to sorting. This minimum memory is insufficient to start even two sort threads. If you have not set **PDQPRIORITY**, check the available memory before you perform a large-scale sort (such as an index build) to make sure that you have enough memory.

Default PSORT_NPROCS values for detached indexes

If the **PSORT_NPROCS** environment variable is set, the database server uses the specified number of sort threads as an upper limit for ordinary sorts. If **PSORT_NPROCS** is not set, parallel sorting does not take place. If you have a single-CPU virtual processor, the database server uses one thread for the sort. If **PSORT_NPROCS** is set to 0, the database server uses three threads for the sort.

Default PSORT_NPROCS values for attached indexes

The default number of threads is different for attached indexes.

If the **PSORT_NPROCS** environment variable is set, you get the specified number of sort threads for each fragment of the index that is being built.

If **PSORT_NPROCS** is not set, or if it is set to 0, you get two sort threads for each fragment of the index unless you have a single-CPU virtual processor. If you have a single-CPU virtual processor, you get one sort thread for each fragment of the index.

For additional information about the **PSORT_NPROCS** environment variable, see your *IBM Informix Administrator's Guide* and your *IBM Informix Performance Guide*.

RTREE_COST_ADJUST_VALUE environment variable

The **RTREE_COST_ADJUST_VALUE** environment variable specifies a coefficient that support functions of user-defined data types can use to estimate the cost of an R-tree index for queries on UDT columns.

```
►►──setenv──RTREE_COST_ADJUST_VALUE──value──────────────────────────►◄
```

value is a floating-point number, where 1 ≤ *value* ≤ 1000, specifying a multiplier for estimating the cost of using an index on a UDT column.

For spatial queries, the I/O overhead tends to exceed by far the CPU cost, so by multiplying the uncorrected estimated cost by an appropriate *value* from this setting, the database server can make better cost-based decisions on how to implement queries on UDT columns for which an R-tree index exists.

SHLIB_PATH environment variable (UNIX)

The **SHLIB_PATH** environment variable tells the shell on HP-UX systems which directories to search for dynamic-link libraries. This is used, for example, with the INTERSOLV DataDirect ODBC Driver. You must specify the full pathname for the directory where you installed the product.

```
                                ┌─:──────────┐
                                ▼            │
►►──setenv──SHLIB_PATH──$PATH:────pathname───┴─────────────────────────────►◄
```

pathname
Specifies the search path for the libraries.

On Solaris systems, set **LD_LIBRARY_PATH**. On AIX systems, set **LIBPATH**.

SRV_FET_BUF_SIZE environment variable

Use the **SRV_FET_BUF_SIZE** environment variable to specify the size of the fetch buffer that the local database server uses in distributed DML transactions across database servers.

```
►►──setenv──SRV_FET_BUF_SIZE──size─────────────────────────────────────────►◄
```

size is a positive integer that is no greater than 1048576 (1 MiB), specifying the size (in bytes) of the fetch buffer that holds data retrieved by a cross-server distributed query.

For example, to set a buffer size to `5,000` bytes on a UNIX system that uses the C shell, set **SRV_FET_BUF_SIZE** by entering the following command:

```
setenv SRV_FET_BUF_SIZE 5000
```

When **SRV_FET_BUF_SIZE** is set to a valid value, the new value overrides the default value (or any previously set value) of **SRV_FET_BUF_SIZE**. The setting takes effect only when it is set in the starting environment of the database server.

When **SRV_FET_BUF_SIZE** is not set, the default setting for the fetch buffer is dependent on row size.

No error is raised if **SRV_FET_BUF_SIZE** is set to a value that is less than the default size, or that is greater than 1048576 (1MiB). If you specify a size for **SRV_FET_BUF_SIZE** that is greater than 1048576, the value is set to 1048576. In older 11.70 releases, up to and including 11.70.xC4, the upper limit is 32767.

A valid **SRV_FET_BUF_SIZE** setting is in effect only in cross-server DML transactions in which the local database server participates as the coordinator or as a subordinate database server.

- It has no effect, however, on queries that access only databases of the local server instance, and it does not affect the size of the fetch buffer in client-to-local-server communication.
- The processing of BYTE and TEXT objects is not affected by the **SRV_FET_BUF_SIZE** setting.

- Setting **SRV_FET_BUF_SIZE** for the environment of the local database server does not reset the fetch buffer size of remote server instances that coordinate or participate in cross-server DML transactions with the local server instance.

The greater the size of the buffer, the more rows can be returned, and the less frequently the local server must wait while the database server returns rows. A large buffer can improve performance when transferring a large amount of data between servers.

STMT_CACHE environment variable

Use the **STMT_CACHE** environment variable to control the use of the shared-statement cache on a session.

This feature can reduce memory consumption and can speed query processing among different user sessions. Valid **STMT_CACHE** values are 1 and 0.

```
►►──setenv──STMT_CACHE──┬─1─┬──────────────────────────────────────────────►◄
                        └─0─┘
```

1 enables the SQL statement cache.

0 disables the SQL statement cache.

Set the **STMT_CACHE** environment variable for applications that do not use the SET STMT_CACHE statement to control the use of the SQL statement cache. By default, a statement cache is disabled, but can be enabled through the STMT_CACHE parameter of the `onconfig.std` file or by the SET STMT_CACHE statement.

This environment variable has no effect if the SQL statement cache is disabled through the configuration file setting. Values set by the SET STMT_CACHE statement in the application override the **STMT_CACHE** setting.

TERM environment variable (UNIX)

The **TERM** environment variable is used for terminal handling. It lets DB-Access (and other character-based applications) recognize and communicate with the terminal that you are using.

```
►►──setenv──TERM──type──────────────────────────────────────────────────────►◄
```

type Specifies the terminal type.

The terminal type specified in the **TERM** setting must correspond to an entry in the `termcap` file or `terminfo` directory.

Before you can set the **TERM** environment variable, you must obtain the code for your terminal from the database administrator.

For example, to specify the vt100 terminal, set the **TERM** environment variable by entering the following command:

```
setenv TERM vt100
```

TERMCAP environment variable (UNIX)

The **TERMCAP** environment variable is used for terminal handling. It tells DB-Access (and other character-based applications) to communicate with the `termcap` file instead of the `terminfo` directory.

```
►►──setenv──TERMCAP──pathname──────────────────────────────────────────►◄
```

pathname
Specifies the location of the `termcap` file.

The `termcap` file contains a list of various types of terminals and their characteristics. For example, to provide DB-Access terminal-handling information, which is specified in the `/usr/informix/etc/termcap` file, enter the following command:

```
setenv TERMCAP /usr/informix/etc/termcap
```

You can use set **TERMCAP** in any of the following ways. If several `termcap` files exist, they have the following (descending) order of precedence:

1. The `termcap` file that you create
2. The `termcap` file that the database server supplies (that is, `$INFORMIXDIR/etc/termcap`)
3. The operating-system `termcap` file (that is, `/etc/termcap`)

If you set the **TERMCAP** environment variable, be sure that the **INFORMIXTERM** environment variable is set to `termcap`.

If you do not set the **TERMCAP** environment variable, the `terminfo` directory is used by default.

TERMINFO environment variable (UNIX)

The **TERMINFO** environment variable is used for terminal handling.

The environment variable is supported only on platforms that provide full support for the `terminfo` libraries that System V and Solaris UNIX systems provide.

```
►►──setenv──TERMINFO──/usr/lib/terminfo────────────────────────────────►◄
```

TERMINFO tells DB-Access to communicate with the `terminfo` directory instead of the `termcap` file. The `terminfo` directory has subdirectories that contain files that pertain to terminals and their characteristics.

To set **TERMINFO**, enter the following command:

```
setenv    TERMINFO   /usr/lib/terminfo
```

THREADLIB environment variable (UNIX)

Use the **THREADLIB** environment variable to compile multithreaded IBM Informix ESQL/C applications. A multithreaded Informix ESQL/C application lets you establish as many connections to one or more databases as there are threads. These connections can remain active while the application program executes.

The **THREADLIB** environment variable indicates which thread package to use when you compile an application. Currently only the Distributed Computing Environment (DCE) is supported.

```
►►──setenv──THREADLIB──DCE─────────────────────────────────────────────►◄
```

The **THREADLIB** environment variable is checked when the `-thread` option is passed to the Informix ESQL/C script when you compile a multithreaded Informix ESQL/C application. When you use the `-thread` option while compiling, the Informix ESQL/C script generates an error if **THREADLIB** is not set, or if **THREADLIB** is set to an unsupported thread package.

TZ environment variable

The **TZ** environment variable is used for setting the time zone. It is used by various time functions to compute times relative to Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time (GMT). The format is specified by the operating system.

```
►►──setenv──TZ──tzn──┬─────┬──hh──┬──────────────────┬──┬─────┬────────►◄
                     ├─ + ─┤      └─:──mm──┬───────┬─┘  └─dzn─┘
                     └─ - ─┘               └─:──ss─┘
```

tzn Three-letter time zone name, such as PST. You must specify the correct offset from local time to UTC (Universal Time Coordinated).

hh A one- or two-digit difference in hours between UTC and local time. Optionally signed.

mm Two-digit difference in minutes between UTC and local time.

ss Two-digit difference in seconds between UTC and local time.

dzn Three-letter daylight-saving-time zone, such as PDT. If daylight saving time is never in effect in the locality, set **TZ** without a value for *dzn*.

For example, if you use Pacific Standard Time with Pacific daylight savings time, set the **TZ** environment variable to `PST8PDT`. For more information on setting the **TZ** environment variable, see your operating system documentation.

USETABLENAME environment variable

The **USETABLENAME** environment variable can prevent users from using a synonym to specify the *table* in ALTER TABLE or DROP TABLE statements. Unlike most environment variables, **USETABLENAME** is not required to be set to a value. It takes effect if you set it to any value, or to no value.

```
►►──setenv──USETABLENAME───────────────────────────────────────────────►◄
```

By default, ALTER TABLE or DROP TABLE statements accept a valid synonym for the name of the *table* to be altered or dropped. (In contrast, RENAME TABLE issues an error if you specify a synonym, as do the ALTER SEQUENCE, DROP SEQUENCE, and RENAME SEQUENCE statements, if you attempt to substitute a synonym for the *sequence* name in those statements.)

If you set **USETABLENAME**, an error results if a synonym is in ALTER TABLE or DROP TABLE statements. Setting **USETABLENAME** has no effect on the DROP VIEW statement, which accepts a valid synonym for the view.

Appendix A. The stores_demo Database

The **stores_demo** database contains a set of tables that describe an imaginary business and many of the examples in the IBM Informix documentation are based on this database.

The **stores_demo** database uses the default (U.S. English) locale and is not ANSI-compliant.

For information about how to create and populate the **stores_demo** database, see the *IBM Informix DB-Access User's Guide*. For information about how to design and implement a relational database, see the *IBM Informix Database Design and Implementation Guide*.

You can see the structure of the tables and their data in the Schema browser in the OpenAdmin Tool (OAT) for Informix.

The stores_demo Database Map

Some of the tables in the **stores_demo** database have relationships between them.

The following illustration displays the joins in the **stores_demo** database between customers, catalog orders, and customer calls. The shading that connects a column in one table to a column with the same name in another table indicates the relationships, or *joins*, between tables.

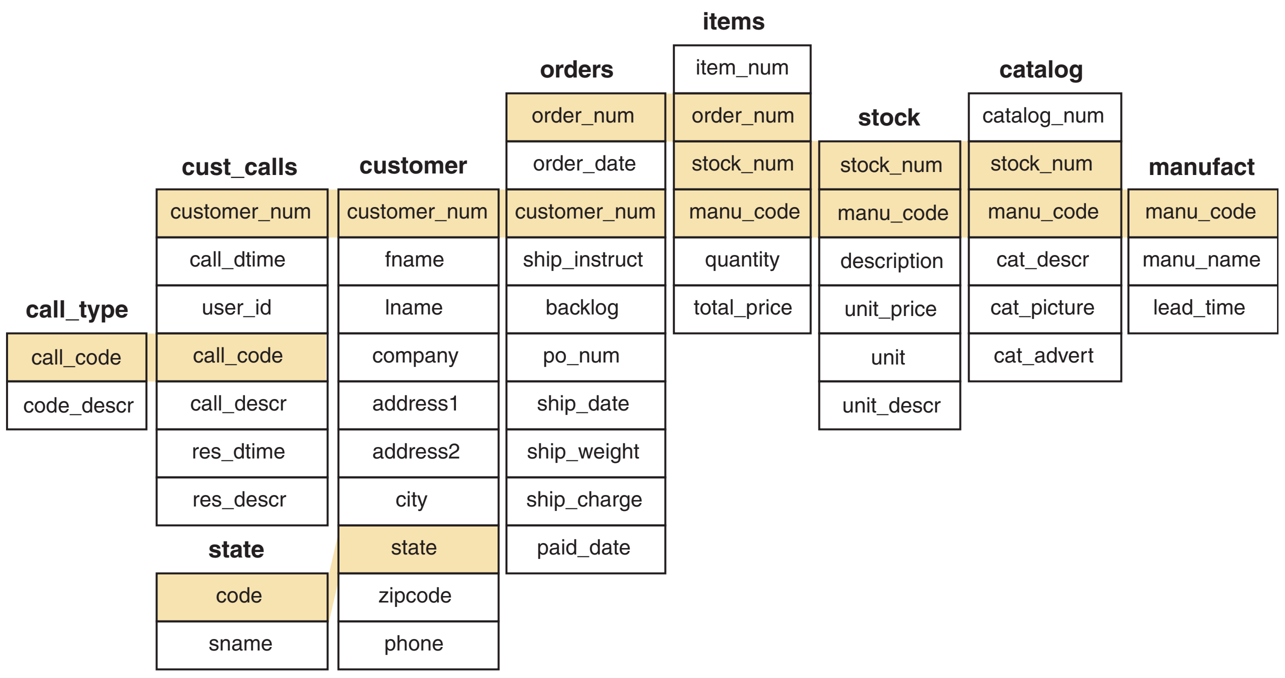


Figure A-1. Joins between customers and catalog orders

The following illustration displays the joins in the **stores_demo** database between customers, electricity meter data, and location. The **Customer_ts_data**, **ts_data**, and **ts_data_location** tables contain time series data. You can prevent the creation of these time series tables when you create the demonstration database.

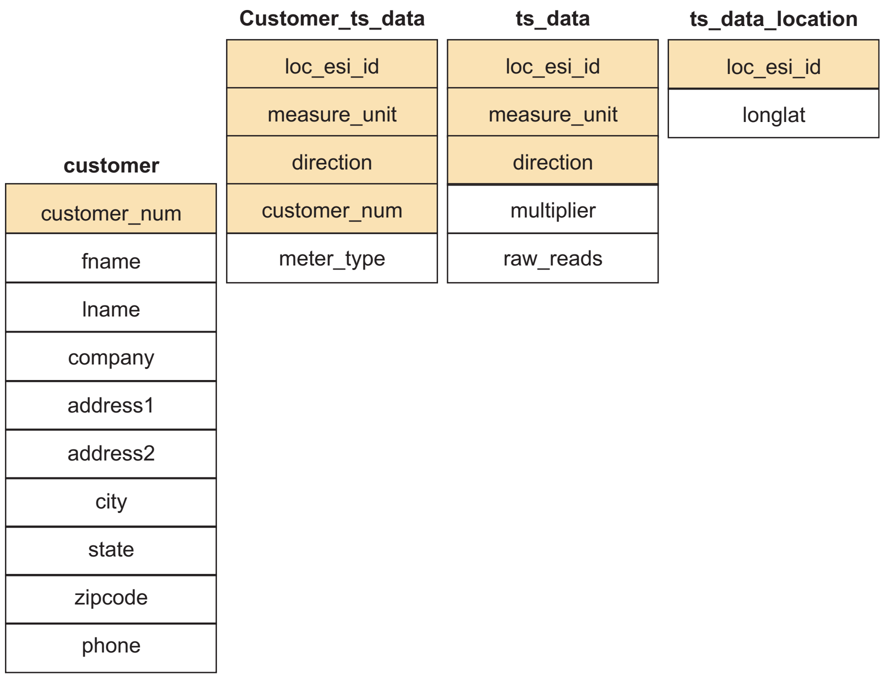


Figure A-2. Joins between customers, electricity usage data, and location

Appendix B. The superstores_demo database

The **superstores_demo** database illustrates an object-relational schema.

SQL files and user-defined routines (UDRs) that are provided with DB-Access let you derive the **superstores_demo** object-relational database.

The **superstores_demo** database uses the default locale and is not ANSI-compliant.

For information about how to create and populate the demonstration databases, including relevant SQL files, see the *IBM Informix DB-Access User's Guide*. For conceptual information about demonstration databases, see the *IBM Informix Database Design and Implementation Guide*.

Structure of the superstores_demo Tables

Although many of the tables in the **superstores_demo** database have the same name as **stores_demo** tables, they are different.

The **superstores_demo** database includes the following tables. The tables are listed alphabetically, not in the order in which they are created.

- **call_type**
- **catalog**
- **cust_calls**
- **customer**
  - **retail_customer**
  - **whlsale_customer**
- **items**
- **location**
  - **location_non_us**
  - **location_us**
- **manufact**
- **orders**
- **region**
- **sales_rep**
- **state**
- **stock**
- **stock_discount**
- **units**

You can see the structure of the tables and their data in the Schema browser in the OpenAdmin Tool (OAT) for Informix.

User-defined routines and extended data types

The **superstores_demo** database uses user-defined routines (UDRs) and extended data types.

A UDR is a routine that you define that can be invoked within an SQL statement or another UDR. A UDR can either return values or not.

The data type system of IBM Informix is an extensible and flexible system that supports the creation of following kinds of data types:

- Extensions of existing data types by, redefining some of the behavior for data types that the database server provides
- Definitions of customized data types by a user

For information about creating and using UDRs and extended data types, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The **superstores_demo** database creates the *distinct* data type, percent, in a UDR, as follows:

```
CREATE DISTINCT TYPE percent AS DECIMAL(5,5);
DROP CAST (DECIMAL(5,5) AS percent);
CREATE IMPLICIT CAST (DECIMAL(5,5) AS percent);
```

The **superstores_demo** database creates the following *named row types*:

- **location** hierarchy:
  - **location_t**
  - **loc_us_t**
  - **loc_non_us_t**
- **customer** hierarchy:
  - **name_t**
  - **customer_t**
  - **retail_t**
  - **whlsale_t**
- **orders** table
  - **ship_t**

location_t definition

```
location_id             SERIAL
loc_type                CHAR(2)
company                 VARCHAR(20)
street_addr             LIST(VARCHAR(25) NOT NULL)
city                    VARCHAR(25)
country                 VARCHAR(25)
```

loc_us_t definition

```
state_code              CHAR(2)
zip                     ROW(code INTEGER, suffix SMALLINT)
phone                   CHAR(18)
```

loc_non_us_t definition

```
province_code           CHAR(2)
zipcode                 CHAR(9)
phone                   CHAR(15)
```

name_t definition

```
first              VARCHAR(15)
last               VARCHAR(15)
```

customer_t definition

```
customer_num                SERIAL
customer_type               CHAR(1)
customer_name               name_t
customer_loc                INTEGER
contact_dates               LIST(DATETIME YEAR TO DAY NOT NULL)
cust_discount               percent
credit_status               CHAR(1)
```

retail_t definition

```
credit_num                  CHAR(19)
expiration                  DATE
```

whlsale_t definition

```
resale_license              CHAR(15)
terms_net                   SMALLINT
```

ship_t definition

```
date                DATE
weight              DECIMAL(8,2)
charge              MONEY(6,2)
instruct            VARCHAR(40)
```

Table Hierarchies

The following illustration shows how the hierarchical tables of the **superstores_demo** database are related. The foreign key and primary relationships between the two tables are indicated by shaded arrows that point from the **customer.custnum** and **customer.loc** columns to the **location.location_id** columns.

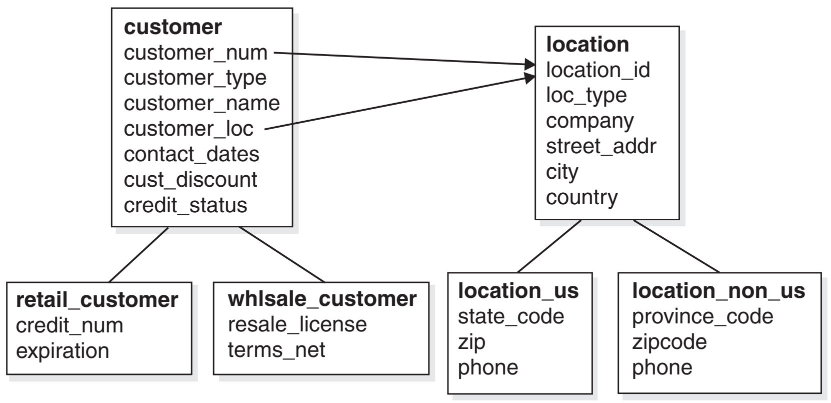


Figure B-1. Hierarchies of superstores_demo Tables

Appendix C. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

IBM and accessibility

For more information about the IBM commitment to accessibility, see the *IBM Accessibility Center* at http://www.ibm.com/able.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number `3.1`) are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number `3` is followed by a series of syntax elements with dotted decimal number `3.1`, all the syntax elements numbered `3.1` are subordinate to the syntax element numbered `3`.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The `*` symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element `*FILE` with dotted decimal number 3 is read as `3 \* FILE`. Format `3* FILE` indicates that syntax element `FILE` repeats. Format `3* \* FILE` indicates that syntax element `* FILE` repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines `5.1*`, `5.1 LASTRUN`, and `5.1 DELETE` mean that if you use more than one of the `LASTRUN` and `DELETE` syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the `%` symbol, that element is defined elsewhere. The string that follows the `%` symbol is the name of a syntax fragment rather than a literal. For example, the line `2.1 %OP1` refers to a separate syntax fragment `OP1`.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the `?` symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the `?` symbol is displayed on the same line as the syntax element (for example, `5? NOTIFY`). If there is more than one syntax element with a dotted decimal number, the `?` symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines `5 ?`, `5 NOTIFY`, and `5 UPDATE`, you know that syntax elements `NOTIFY` and `UPDATE` are optional; that is, you can choose one or none of them. The `?` symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the `!` symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a `!` symbol. For example, if you hear the lines `2? FILE`, `2.1! (KEEP)`, and `2.1 (DELETE)`, you know that `(KEEP)` is the default option for the `FILE` keyword. In this example, if you include the `FILE` keyword but do not specify an option, default option `KEEP` is applied. A default option also applies to the next higher dotted decimal number. In this example, if the `FILE` keyword is omitted, default `FILE(KEEP)` is used. However, if you hear the lines `2? FILE`, `2.1`, `2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP only applies to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the `*` symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*`, `3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. This material may be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

Privacy policy considerations

Trademarks

Index

Special characters

(_), underscore
    in SQL identifiers 3-37
(;), semicolon
    list separator 3-54, 3-63
(:), colon
    cast (::) operator 2-53, 2-55
    DATETIME delimiter 2-12
    INTERVAL delimiter 2-19
    list separator 3-30, 3-38, 3-54, 3-59, 3-63
(!=), not equal to
    relational operator 2-55
(/), slash
    DATE separator 2-12, 2-44, 3-23
    division operator 2-41, 2-55
    pathname delimiter 3-5, 3-28, 3-59
(()), parentheses
    delimiters in expressions 2-45
($), dollar sign
    currency symbol 2-24, 3-27
    pathname indicator 3-63
(\ ), backslash
    invalid as delimiter 3-24
    pathname delimiter 3-7, 3-55
([ ]), brackets
    substring operator 2-8, 2-55
(%), percentage
    DBTIME escape symbol 3-32
    pathname indicator 3-30
(>), greater than
    angle (< >) brackets 2-8
    relational operator 1-6, 2-55
(<), less than
    angle (< >) brackets 2-8
    relational operator 2-55, 3-24
(|), vertical bar
    absolute value delimiter 2-19
    concatenation (||) operator 2-55
    field delimiter 3-24
(#), sharp
    comment indicator 3-2
('), single quotation
    string delimiter 3-27
('), single quotation symbols
    string delimiter 3-37
("), double quotation marks
    string delimiter 2-22
("), double quotation symbols
    delimited SQL identifiers 3-37
    string delimiter 2-1, 2-25, 2-32
({ }), braces
    collection delimiters 2-22, 2-25
    pathname delimiters 3-4
(-), hyphen
    DATE separator 3-23
    DATETIME delimiter 2-12
    INTERVAL delimiter 2-19
    subtraction operator 2-41, 2-55
    symbol in syscolauth 1-1, 1-16
    symbol in sysfragauth 1-28
    symbol in systabauth 1-51
(-), hyphen *(continued)*
    unary operator 2-42, 2-55
(,), comma
    decimal point 3-27
    list separator 2-25, 2-28, 3-30
    thousands separator 2-24
(.), period
    DATE separator 3-23
    DATETIME delimiter 2-12
    decimal point 2-16, 2-24, 3-27
    execution symbol 3-2
    INTERVAL delimiter 2-19
    membership operator 2-55
    nested dot notation 2-48
(), blank space
    DATETIME delimiter 2-12
    INTERVAL delimiter 2-19
    padding CHAR values 2-10
    padding VARCHAR values 2-35
(*), asterisk
    multiplication operator 2-7, 2-41, 2-45, 2-55
    systabauth value 1-1, 1-51
    wildcard symbol 1-15, 1-63
(+), plus sign
    addition operator 2-41, 2-55
    truncation indicator 3-43
    unary operator 2-55
(=), equality
    assignment operator 3-7
    relational operator 1-15, 2-7, 2-11, 2-55
(~), tilde
    pathname indicator 3-5
' VERSION' table 1-52

A

Abbreviated year values 2-12, 3-20, 3-22, 3-23, 3-32
ACCESS keyword 2-40
Access method
    B-tree 1-10, 1-33, 3-36
    built-in 1-10
    primary 1-10, 1-51
    R-Tree 3-36
    secondary 1-10, 1-21, 1-35, 2-26
    sysams data 1-10
    sysindices data 1-35
    sysopclasses data 1-39
    systabamdata data 1-51
ACCESS_METHOD keyword 1-10
Accessibility C-1
    dotted decimal format of syntax diagrams C-1
    keyboard C-1
    shortcut keys C-1
    syntax diagrams, reading in a screen reader C-1
Addition (+) operator 2-41, 2-55
Administrative listener port 3-48
Aggregate functions 2-33
    built-in 2-22, 2-25, 2-32
    no BYTE argument 2-8
    no collection arguments 2-22, 2-25, 2-32
    sysaggregates data 1-9

Aggregate functions *(continued)*
    user-defined 1-9
AIX operating system 3-42, 3-59
Alias of a table 1-1
Alignment of data type 1-61
Alignment of data types 1-14
ALL operator 2-55
Alter privilege 1-1, 1-51, 1-64
ALTER SEQUENCE statement 3-73
ALTER TABLE statement
    casting effects 2-50
    changing data types 2-1
    lock mode 3-40
    next extent size 1-6
    SERIAL columns 2-30
    SERIAL8 columns 2-31
    synonyms 3-73
am_beginscan() function 1-10
am_close() function 1-10
am_getnext() function 1-10
am_insert() function 1-10
am_open() function 1-10
AND operator 1-15, 2-55
ANSI compliance
    -ansi flag 3-19
    DATETIME literals 3-32
    DBANSIWARN environment variable 3-19
    DECIMAL range 2-16
    DECIMAL(p) data type 2-15
    Information Schema views 1-62
    isolation level 1-65
    public synonyms 1-50, 1-52
ANSI data types 2-6
ANSIOWNER environment variable 3-16
ANY operator 2-55
Arabic locales 2-9
Archiving
    setting DBREMOTECMD 3-30
Arithmetic
    DATE operands 2-12, 2-43
    DATETIME operands 2-42
    integer operands 2-7, 2-19, 2-33
    INTERVAL operands 2-19, 2-43
    operators 2-55
    string operands 2-9
    time operands 2-41
AS keyword 2-53
assign() support function 2-46
AT keyword 2-22
Attached index 3-36
Attached indexes 1-31, 3-22, 3-69
Audit Analysis officer 3-57
Authorization identifier 1-58, 1-65
AUTO_STAT_MODE configuration parameter 1-24, 1-29
AUTO_STAT_MODE session environment setting 1-24, 1-29

B

B-tree access method 1-10, 1-33, 3-36
B-tree index 1-31
Backslash (\ ) symbol 3-24
Backup
    file prefix 3-46
Bandwidth 3-46
BETWEEN operator 2-55
BIGINT data type 2-6
    coltype code 1-17
BIGINT data type *(continued)*
    length (syscolumns) 1-20
BIGSERIAL data type 2-6
    coltype code 1-17
    length (syscolumns) 1-20
bin subdirectory 3-4
Binding style 1-65
BLOB data type
    casting unavailable 2-7
    defined 2-7
    inserting data 2-7
    syscolattribs data 1-16
Blobspaces
    defined 2-40
    names 3-37
    sysblobs data 1-14
BOOLEAN data type
    defined 2-8
Boolean expression
    with BOOLEAN data type 2-8
    with BYTE data type 2-8
Boolean expression with TEXT data type 2-33
Borland C compiler 3-50
Bourne shell 3-2, 3-3
Bracket ([ ]) symbols 2-33
brackets substring 2-33
Buffers
    BYTE or TEXT storage (DBBLOBBUF) 3-20
    fetch buffer (FET_BUFFER_SIZE) 3-39
    fetch buffer (SRV_FET_BUFFER_SIZE) 3-70
    floating-point display (DBFLTMASK) 3-25
    network buffer (IFX_NETBUF_SIZE) 3-44
    private network buffer pool 3-44
Built-in access method 1-10
Built-in aggregates 1-9, 2-22, 2-25, 2-32
Built-in casts 1-14, 2-50
Built-in data types
    casts 2-50, 2-54
    listed 2-37
    syscolumns.coltype code 1-17
    sysdistrib.type code 1-24
    sysxtdtypes data 1-61
Built-in opaque data types 2-53
BY clause 2-33
BY keyword 2-8, 2-33
BY ORDER 2-33
BYTE data type
    casting to BLOB 2-8
    coltype code 1-17
    defined 2-8
    increasing buffer size 3-20
    inserting values 2-8
    restrictions
        in Boolean expression 2-8
        systables.npused 1-52
        with GROUP BY 2-8
        with LIKE or MATCHES 2-8
        with ORDER BY 2-8
    selecting from BYTE columns 2-8
    setting buffer size 3-20
    sysblobs data 1-14
    syscolumns data 1-20
    sysfragments data 1-31

C

C compiler
  default name 3-50
  INFORMIXC setting 3-50
  thread package 3-72
C shell 3-2
  .cshrc file 3-3
  .login file 3-3
C++ map file 3-54
CARDINALITY() function 2-22, 2-25, 2-32
Cascading deletes 1-44
Case-insensitive databases 1-7, 2-25, 2-26
Cast (::) operator 2-53, 2-55
CAST AS keywords 2-53
casting to CLOB 2-33
Casts 2-50, 2-54
  built-in 1-14, 2-50, 2-53
  distinct data type 2-53
  explicit 1-14, 2-53
  from BYTE to BLOB 2-8
  implicit 1-14, 2-53
  rules of precedence 2-53
  syscasts data 1-14
  user-defined (UDCs) 1-14
Casts from TEXT 2-33
CHAR data type
  built-in casts 2-52
  collation 2-9, 2-10
  defined 2-9
  nonprintable characters 2-10
  storing numeric values 2-9
CHARACTER data type 2-10
Character data types
  Boolean comparisons 2-35
  casting between 2-50
  data strings 2-1
  listed 2-37
Character string
  CHAR data type 2-9
  CHARACTER VARYING data type 2-10
  CLOB data type 2-11
  DATETIME literals 2-12, 2-44, 3-32
  INTERVAL literals 2-19
  LVARCHAR data type 2-23
  NCHAR data type 2-25
  NVARCHAR data type 2-26
  VARCHAR data type 2-35
  with DELIMIDENT set 3-37
CHARACTER VARYING data type
  defined 2-10
Character-based applications 3-56, 3-71
Check constraints
  creation-time value 3-22, 3-24
  syschecks data 1-15
  syscheckudrdep data 1-15
  syscoldepend data 1-17
  sysconstraints data 1-22
chkenv utility 3-2
  error message 3-4
  syntax 3-4
Chunks 2-40
CLIENT_LOCALE environment variable 3-23
Client/server
  DataBlade API 2-40
  default database 3-54
  INFORMIXSQLHOSTS environment variable 3-55
  shared memory communication segments 3-55
Client/server *(continued)*
  stacksize for client session 3-56
CLOB data type
  casting unavailable 2-11
  code-set conversion 2-11
  collation 2-11
  defined 2-11
  inserting data 2-11
  multibyte characters 2-11
  syscolattribs data 1-16
CLOB TEXT 2-33
CLOSE statement 3-62
Clustering 1-10, 1-31, 1-35
CMCONFIG environment variable 3-17
Code sets
  conversion 3-9
  East Asian 2-10, 3-32
  EBCDIC 1-65
  ISO 8859-1 1-26
Collation 2-33
  CHAR data type 2-9, 2-10
  CLOB data type 2-11
  GL_COLLATE table 1-52
  NCHAR data type 2-25
  NVARCHAR data type 2-26
  server_attribute data 1-65
Collection data type
  casting matrix 2-54
  defined 2-47
  empty 2-47
  LIST 2-22
  MULTISET 2-25
  SET 2-32
  sysattrtypes data 1-12
  sysxtddesc data 1-60
  sysxtdtypes data 1-60, 1-61
COLLECTION data type
  coltype code 1-17
collection delimiters 2-32, 2-47
Colon
  cast (::) operator 2-53
  DATETIME delimiter 2-12
  INTERVAL delimiter 2-19
Color and intensity screen attributes 3-56
Column-level privileges
  systabauth data 1-1
  systabauth table 1-51
Columns
  changing data type 2-1, 2-50
  constraints (sysconstraints) 1-22
  default values (sysdefaults) 1-22
  hashed 1-31
  in superstores_demo database B-1
  inserting BLOB data 2-7
  range of values 1-21
  syscolumns data 1-17
columns Information Schema view 1-62
Combine function 1-9
Comment indicator 3-2
Comment lines 3-2
Committed read 1-65
Communications support module 3-51
Commutator function 1-42
Compiling
  ESQL/C programs 3-17
  INFORMIXC setting 3-50
  JAVA_COMPILER setting 3-58

Compiling *(continued)*
  multithreaded ESQL/C applications 3-72
Complex data type 2-46, 2-49
  collection types 2-47
  ROW types 2-48
  sysattrtypes data 1-12
Compliance
  ANSI/ISO standard for SQL 1-62, 3-19
  sql_languages.conformance 1-65
  X/Open CAE standards 1-62
  XPG4 standard 1-64
compliance with standards xiii
Composite index 1-33
Concatenation (||) operator 2-55
concsm.cfg file 3-51
Confidence level 1-29
Configuration file
  .cshrc file 3-3
  .informix 3-2, 3-4, 3-38, 3-40
  .login file 3-3
  .profile file 3-3
  for communications support module 3-51
  for connectivity 3-49, 3-54, 3-55
  for database servers 3-38, 3-60
  for High-Performance Loader 3-64
  for MaxConnect 3-49
  for terminal I/O 3-56
Configuration parameters
  DBSPACETEMP 3-30
  DEF_TABLE_LOCKMODE 3-40
  DIRECTIVES 3-40
  DISABLE_B162428_XA_FIX 3-48
  EXT_DIRECTIVES 1-24, 3-41
  MITRACE_OFF 1-55, 1-56
  OPT_GOAL 3-63
  OPTCOMPIND 3-61
  RESIDENT 3-42
  shared memory base 3-48
  SQL_LOGICAL_CHAR 1-52
  STACKSIZE 3-56
  STMT_CACHE 3-71
  USEOSTIME 2-12
CONNECT DEFAULT statement 3-54
Connect privilege 1-6, 1-58
CONNECT statement 3-28, 3-52, 3-54
Connections
  INFORMIXCONRETRY environment variable 3-51
  INFORMIXCONTIME environment variable 3-52
  INFORMIXSERVER environment variable 3-54
Connectivity information 3-48, 3-55
Constraints
  check
    creation-time value 3-24
    syschecks data 1-15
    syscheckudrdep data 1-15
    syscoldepend data 1-17
  column
    sysconstraints data 1-22
  not null
    collection data types 2-25, 2-32, 2-47
  NOT NULL
    collection data types 2-22
    syscoldepend data 1-17
    syscolumns data 1-17
    sysconstraints data 1-22
  object mode 1-38
Constraints *(continued)*
  primary key
    sysconstraints data 1-22
    sysreferences data 1-44
    unique SERIAL values 2-30
    unique SERIAL8 values 2-30
  referential
    sysconstraints data 1-22
    sysreferences data 1-44
  table
    sysconstraints data 1-22
  unique
    sysconstraints data 1-22
    sysviolations data 1-59
  violations 1-59
Constructors 2-32, 2-47
Converting data types
  DATE and DATETIME 2-52
  INTEGER and DATE 2-52
  number and string 2-52
  number to number 2-51
  retyping a column 2-50
CPFIRST environment variable 3-17
CPU cost 3-69
CREATE ACCESS_METHOD statement 1-10
CREATE CAST statement 1-14, 2-52
CREATE DATABASE statement 3-28
CREATE DISTINCT TYPE statement 1-61, 2-17, B-2
CREATE EXTERNAL TABLE statement 1-27, 1-28
CREATE FUNCTION statement 1-45
CREATE IMPLICIT CAST statement B-2
CREATE INDEX statement 1-33, 1-35, 1-52, 3-36
  storage options 3-36
CREATE OPAQUE TYPE statement 2-26
CREATE OPERATOR CLASS statement 1-39
CREATE PROCEDURE statement 1-45, 3-59
CREATE ROLE statement 1-45
CREATE ROUTINE FROM statement 1-45, 3-59
CREATE ROW TYPE statement 1-17, 2-27
CREATE SCHEMA statement 1-1
CREATE SEQUENCE statement 1-49
CREATE SYNONYM statement 1-50
CREATE TABLE statement
  assigning data types 2-1
  default lock mode 3-40
  default privileges 3-59
  SET constructor 2-32
  typed tables 2-27
CREATE TEMP TABLE statement 3-30
CREATE TRIGGER statement 1-57
CREATE VIEW statement 1-1, 1-58
CREATE XADATASOURCE statement 1-59
CREATE XADATASOURCETYPE statement 1-60
Currency symbol 2-24, 3-27
Current date 1-22, 3-20
CURRENT keyword 2-41

D

Data corruption 1-6, 1-16
Data dependencies
  syscheckudrdep data 1-15
  syscoldepend data 1-17
  sysdepend data 1-23
Data dictionary 1-1
Data distributions 1-6, 1-24, 3-35
Data integrity 1-65

Data pages 1-16, 1-33, 1-52
data type collation 2-33
data type restrictions 2-33
data type restrictions in Boolean expression 2-33
data type UPDATE statements 2-33
Data types
  ANSI 2-6
  approximate 1-64
  BIGINT 2-6
  BIGSERIAL 2-6
  BLOB 2-7
  BOOLEAN 2-8
  BYTE 2-8
  casting 2-50, 2-54
  CHAR 2-9
  CHARACTER 2-10
  CHARACTER VARYING 2-10
  classified by category 2-1
  CLOB 2-11
  collection 2-47
  complex 2-46
  conversion 2-50
  DATE 2-12
  DATETIME 2-12
  DEC 2-15
  DECIMAL 2-15
  distinct 2-49
  DISTINCT 2-17
  DOUBLE PRECISION 2-18
  exact numeric 1-64
  extended 2-46
  fixed point 2-16
  FLOAT 2-18
  floating-point 2-15, 2-18, 2-33
  IDSSECURITYLABEL 2-18, 2-37
  inheritance 2-27
  INT 2-19
  INT8 2-19
  INTEGER 2-19
  internal 2-1
  INTERVAL 2-19
  length (syscolumns) 1-20
  LIST 2-22
  LVARCHAR 2-23
  MONEY 2-24
  MULTISET 2-25
  named ROW 2-27
  NCHAR 2-25
  NUMERIC 2-26
  NVARCHAR 2-26
  opaque 2-49
  OPAQUE 2-26
  Opaque data types
    smart large objects 2-40
  REAL 2-27
  ROW 2-27, 2-28
  sequential integer 2-30
  SERIAL 2-30
  SERIAL8 2-30
  SET 2-32
  simple large object 2-40
  SMALLFLOAT 2-33
  SMALLINT 2-33
  smart large object 2-40
  summary list 2-1
  unique numeric value 2-30
  unnamed ROW 2-28
Data types *(continued)*
  VARCHAR 2-35
Data-type promotion 2-37
Database identifiers 3-37
Database server administrator (DBSA) 1-1
Database Server Administrator (DBSA) 3-57
Database servers
  attributes in Information Schema view 1-65
  code set 1-65
  default connection 3-54
  default isolation level 1-65
  optimizing queries 3-63
  pathname for 3-28
  remote 3-39
  role separation 3-57
  server name 1-22, 3-28
DATABASE statement 3-28
Databases
  data types 2-1
  Databases
    superstores_demo B-1
  demonstration databases
    superstores_demo B-1
  identifiers 3-37
  joins in stores_demo A-1
  object-relational B-1
  objects, sysobjstate data 1-38
  privileges 1-58
  stores_demo A-1
  superstores_demo
    demonstration database B-1
  syscrd 1-1
  sysmaster 1-1
  sysutils 1-1
  sysuuid 1-1
DataBlade modules
  Client and Server API 2-40
  data types (sysbuiltintypes) 1-1
  trace messages (systracemsgs) 1-55, 1-56
  user messages (syserrors) 1-26
DATE data type
  abbreviated year values 3-20
  casting to integer 2-52
  coltype code 1-17
  converting to DATETIME 2-52
  defined 2-12
  display format 3-23
  in expressions 2-41, 2-43
  international date formats 2-12
  source data 2-43
DATE() function 2-44, 3-23
DATETIME data type
  abbreviated year values 3-20
  coltype code 1-17
  converting to DATE 2-52
  defined 2-12
  display format 3-32
  EXTEND function 2-43
  extending precision 2-42
  field qualifiers 2-12
  in expressions 2-41, 2-45
  international formats 2-12, 2-19
  length (syscolumns) 1-20
  literal values 2-12
  localized values 2-12
  precision and size 2-12
  source data 2-44

DATETIME data type *(continued)*
  two-digit year values and DBDATE variable 2-12
  year to fraction example 2-12
DAY keyword
  DATETIME qualifier 2-12
  INTERVAL qualifier 2-19
  UNITS operator 2-12, 2-44
DB-Access utility 1-6, 1-63, 3-6, 3-25, 3-28, 3-32, 3-54
DBA privilege 1-26, 1-55, 1-56, 1-58
DBA routines 1-42
DBACCNOIGN environment variable 3-18, 3-19
DBANSIWARN environment variable 3-19
DBBLOBBUF environment variable 3-20
DBCENTURY environment variable
  defined 3-20
  effect on functionality of DBDATE 3-23
  expanding abbreviated years 2-12, 3-21
DBDATE environment variable 2-12, 3-23
DBDELIMITER environment variable 3-24
DBEDIT environment variable 3-25
dbexport utility 3-24
DBFLTMASK environment variable 3-25
DBLANG environment variable 3-26
dbload utility 2-7, 2-8, 2-33, 3-24
DBMONEY environment variable 2-24, 3-27
DBONPLOAD environment variable 3-28
DBPATH environment variable 3-28
DBPRINT environment variable 3-30
DBREMOTECMD environment variable 3-30
dbschema utility 1-42
DBSECADM role 2-18, 2-37
dbservername.cmd batch file 3-8
dbspace
  for BYTE or TEXT values 1-14
  for system catalog 1-1
  for table fragments 1-28
  for temporary tables 3-30
  name 3-37
DBSPACETEMP configuration parameter 3-30
DBSPACETEMP environment variable 3-30
DBTEMP environment variable 3-32
DBTIME environment variable 2-12, 3-32
DBUPSPACE environment variable 3-35
DEC data type 2-15
DECIMAL data type
  built-in casts 2-51, 2-52
  coltype code 1-17
  defined 2-15
  disk storage 2-16
  display format 3-25, 3-27
  fixed point 2-16
  floating point 2-15
  length (syscolumns) 1-20
Decimal digits, display of 3-25
Decimal point
  DBFLTMASK setting 3-25
  DBMONEY setting 3-27
  DECIMAL radix 2-16
Decimal separator 3-27
DECLARE statement 3-62
DECRYPT_BINARY function 2-11
DECRYPT_CHAR function 2-11
DEF_TABLE_LOCKMODE configuration parameter 3-40
Default database locale 1-7
DEFAULT_ATTACH environment variable 3-36
Defaults
  C compiler 3-50
Defaults *(continued)*
  century 3-20, 3-32
  CHAR length 2-9
  character set for SQL identifiers 3-37
  compilation order 3-17
  configuration file 3-60
  connection 3-54
  data type 2-29
  database server 3-28, 3-54
  DATE display format 2-12
  DATE separator 3-23
  DATETIME display format 2-12
  DECIMAL precision 2-15
  disk space for sorting 3-35
  fetch buffer size 3-39
  heap size 3-58
  index storage location 3-36
  isolation level 1-65
  join method 3-61
  level of parallelism 3-64
  lock mode 3-40
  message directory 3-26
  MONEY scale 2-24
  operator class 1-10, 1-39
  printing program 3-30
  query optimizer goal 3-63
  sysdefaults.default 1-22
  table privileges 3-59
  temporary dbspace 3-30
  terminfo direcotry 3-72
  text editor 3-25
DEFINE statement of SPL 2-30
defined Data types 2-33
Delete privilege 1-28, 1-51, 3-59
DELETE statement 1-59
DELETE statements 1-6
Delete trigger 1-57
DELIMIDENT environment variable 3-37
Delimited identifiers 3-37
Delimiter
  for DATETIME values 2-12
  for fields 3-24
  for identifiers 3-37
  for INTERVAL values 2-19
demonstration databases
  stores_demo A-1
Demonstration databases
  tables B-1
Descending index 1-33
DESCRIBE statement 3-47
Describe-for-updates 3-47
destroy() support function 2-46
Detached index 3-36
Deutsche mark (DM) currency symbol 3-27
Diagnostics table 1-59
DIRECTIVES configuration parameter 3-40
Directives for query optimization 3-40, 3-61, 3-63
Disabilities, visual
  reading syntax diagrams C-1
Disability C-1
Disabled database objects 1-59
Disk space
  for data distributions 3-35
  for temporary data 3-30
Distinct data types
  casts 2-53
  sysxtdtypes data 1-61

DISTINCT data types
  defined 2-17
  sysxtddesc data 1-60
  sysxtdtypes data 1-61, 2-17
Distributed Computing Environment (DCE) 3-72
Distributed queries 2-46, 3-39
Dollar ($) sign 2-24, 3-27
Dotted decimal format of syntax diagrams C-1
double (C) data type 2-18
Double-precision floating-point number 2-18
DROP CAST statement B-2
DROP DATABASE statement 3-28
DROP FUNCTION statement 1-42
DROP INDEX statement 1-52
DROP PROCEDURE statement 1-42
DROP ROUTINE statement 1-42
DROP ROW TYPE statement 2-27
DROP SEQUENCE statement 3-73
DROP TABLE statement 3-73
DROP TYPE statement 2-17, 2-26
DROP VIEW statement 1-63, 3-73

E

EBCDIC collation 1-65
Editor, DBEDIT setting 3-25
EMACS text editor 3-25
Empty set 2-47
ENCRYPT_DES function 2-11
ENCRYPT_TDES function 2-11
Enterprise Replication 1-1
env utility 3-4
ENVIGNORE environment variable
  defined 3-2, 3-38
  relation to chkenv utility 3-4
Environment configuration file
  debugging with chkenv 3-4
  setting environment variables in UNIX 3-2
Environment variables
  ANSIOWNER 3-16
  CLIENT_LOCALE 3-23
  CMCONFIG 3-17
  Colon
    pathname separator 3-59
  command-line utilities 3-6
  CPFIRST 3-17
  DBACCNOIGN 3-18, 3-19
  DBANSIWARN 3-19
  DBBLOBBUF 3-20
  DBCENTURY 3-20
  DBDATE 2-12, 3-23
  DBDELIMITER 3-24
  DBEDIT 3-25
  DBFLTMASK 3-25
  DBLANG 3-26
  DBMONEY 2-24, 3-27
  DBONPLOAD 3-28
  DBPATH 3-28
  DBPRINT 3-30
  DBREMOTECMD 3-30
  DBSPACETEMP 3-30
  DBTEMP 3-32
  DBTIME 2-12, 3-32
  DBUPSPACE 3-35
  DEFAULT_ATTACH 3-36
  DELIMIDENT 3-37
  displaying current settings 3-4, 3-7

Environment variables *(continued)*
  ENVIGNORE 3-38
  FET_BUF_SIZE 3-39
  GL_DATE 2-12, 3-22
  GL_DATETIME 2-12, 3-22
  how to set
    in Bourne shell 3-3
    in C shell 3-3
    in Korn shell 3-3
  how to set in Bourne shell 3-3
  how to set in Korn shell 3-3
  IFMXMONGOAUTH 3-39
  IFX_DEF_TABLE_LOCKMODE 3-40
  IFX_DIRECTIVES 3-40
  IFX_EXTDIRECTIVES 1-24, 3-41
  IFX_LARGE_PAGES 3-42
  IFX_LOB_XFERSIZE 3-43
  IFX_LONGID 3-43
  IFX_NETBUF_PVTPOOL_SIZE 3-44
  IFX_NETBUF_SIZE 3-44
  IFX_NO_SECURITY_CHECK 3-45
  IFX_NO_TIMELIMIT_WARNING 3-45
  IFX_NODBPROC 3-45
  IFX_NOT_STRICT_THOUS_SEP 3-46
  IFX_ONTAPE_FILE_PREFIX 3-46
  IFX_PAD_VARCHAR 3-46
  IFX_UNLOAD_EILSEQ_MODE 3-47
  IFX_UPDDESC 3-47
  IFX_XASTDCOMPLIANCE_XAEND 3-48
  IFX_XFER_SHMBASE 3-48
  IMCADMIN 3-48
  IMCCONFIG 3-49
  IMCSERVER 3-49
  INF_ROLE_SEP 3-57
  INFORMIXC 3-50
  INFORMIXCMCONUNITNAME 3-51
  INFORMIXCMNAME 3-50
  INFORMIXCONCSMCFG 3-51
  INFORMIXCONRETRY 3-51
  INFORMIXCONTIME 3-52
  INFORMIXCPPMAP 3-54
  INFORMIXDIR 3-54
  INFORMIXSERVER 3-54
  INFORMIXSHMBASE 3-55
  INFORMIXSQLHOSTS 3-55
  INFORMIXSTACKSIZE 3-56
  INFORMIXTERM 3-56
  INTERACTIVE_DESKTOP_OFF 3-57
  JAR_TEMP_PATH 3-58
  JAVA_COMPILER 3-58
  JVM_MAX_HEAP_SIZE 3-58
  LD_LIBRARY_PATH 3-59
  LIBPATH 3-59
  limitations 3-1
  listed by topic 3-9
  manipulating in Windows environments 3-6
  modifying settings 3-4
  NODEFDAC 3-59
  ONCONFIG 3-60
  ONINIT_STDOUT 3-60
  OPT_GOAL 3-63
  OPTCOMPIND 3-61
  OPTMSG 3-62
  OPTOFC 3-62
  overriding a setting 3-2, 3-38
  PATH 3-63

Environment variables *(continued)*
  Pathname
    for client or shared libraries 3-59
  PDQPRIORITY 3-64
  PLCONFIG 3-64
  PLOAD_LO_PATH 3-65
  PLOAD_SHMBASE 3-65
  PSM_ACT_LOG 3-66
  PSM_DBS_POOL 3-66
  PSM_DEBUG 3-67
  PSM_DEBUG_LOG 3-67
  PSM_LOG_POOL 3-68
  PSORT_DBTEMP 3-68
  PSORT_NPROCS 3-68
  RTREE_COST_ADJUST_VALUE 3-69
  rules of precedence in UNIX 3-5
  rules of precedence in Windows 3-8
  scope of reference 3-6
  setting 3-6
    at the command line 3-2
    in a configuration file 3-2
    in a login file 3-2
    in a shell file 3-3
    in Windows environments 3-6
    with the System applet 3-6
  setting in autoexec.bat 3-7
  SHLIB_PATH 3-70
  SRV_FET_BUF_SIZE 3-70
  standard UNIX system 3-1
  STMT_CACHE 3-71
  TERM 3-71
  TERMCAP 3-72
  TERMINFO 3-72
  THREADLIB 3-72
  types of 3-1
  unsetting 3-4, 3-7, 3-37
  USE_DTENV 2-12
  USETABLENAME 3-73
  view current setting 3-4
  where to set 3-3
equal() support function 2-46
Equality (=) operator 2-11
Era-based dates 3-32
Error message files 3-26
esql command 3-17, 3-50
ESQL/C
  DATETIME routines 3-32
  esqlc command 3-17
  long identifiers 3-43
  message chaining 3-62
  multithreaded applications 3-72
  program compilation order 3-17
Exact numeric data types 1-64
Executable programs 3-63
Execute privilege 1-40, 3-59
explain output file 3-35
Explicit cast 1-14, 2-53
Explicit pathnames 3-7, 3-29
Explicit temporary tables 3-30
Exponent 2-16
Exponential notation 2-15
export utility 3-3
export_binary() support function 2-46
export() support function 2-46
Expression-based fragmentation 1-29, 1-31, 3-22, 3-24
EXT_DIRECTIVES configuration parameter 1-24, 3-41
EXTEND function 2-43
Extended data types 1-61, 2-46, B-2
Extensible Markup Language (XML) 2-11
Extension checking (DBANSIWARN) 3-19
Extents, changing size 1-6
External database 1-50
External directives for query optimization 3-41
External routines 1-42
External tables
  sysextcols data 1-27
  sysextdfiles data 1-27
  sysexternal data 1-28
  systables data 1-52
External view 1-50
extspace 1-10

F

FALSE setting
  BOOLEAN value 2-8
Farsi locales 2-9
FET_BUF_SIZE environment variable 3-39
Fetch buffer 3-39
Fetch buffer size 3-39
FETCH statement 3-62
Field delimiter
  DBDELIMITER 3-24
  Statements of SQL
    LOAD 3-24
    UNLOAD 3-24
  Utilities
    dbexport 3-24
Field of a ROW data type 2-48
Field qualifier
  DATETIME values 2-12
  EXTEND function 2-43
  INTERVAL values 2-19
Fields of a ROW data type 2-48
File extensions
  .a 3-43
  .cfg 3-51
  .cmd 3-8
  .ec 3-17
  .ecp 3-17
  .iem 3-26
  .jar 3-58
  .rc 3-2, 3-5, 3-38, 3-40
  .so 3-43
  .sql 1-63, 3-28, 3-37
  .std 3-60, 3-71
Files
  environment configuration files 3-4
  installation directory 3-54
  permission settings 3-2
  shell 3-3
  temporary 3-30, 3-32, 3-68
  temporary for SE 3-32
  termcap, terminfo 3-56, 3-72
FILETOBLOB function 2-7
FILETOCLOB function 2-11
Filtering mode 1-38, 1-59
Finalization function 1-9
Fixed point decimal 2-16, 2-24, 3-27
Fixed-length opaque data types 1-17
Fixed-length UDT 1-61
FLOAT data type
  built-in casts 2-51, 2-52
  coltype code 1-17

FLOAT data type *(continued)*
  defined 2-18
  display format 3-25, 3-27
Floating-point decimal 2-15, 2-18, 2-33, 3-25
Formatting
  DATE values with DBDATE 3-23
  DATE values with GL_DATE 3-32
  DATETIME values with DBTIME 3-32
  DATETIME values with GL_DATETIME 3-32
  DATETIME values with USE_DTENV 3-32
  DECIMAL(p) values with DBFLTMASK 3-25
  FLOAT values with DBFLTMASK 3-25
  MONEY values with DBMONEY 3-27
  SMALLFLOAT values with DBFLTMASK 3-25
Formatting mask
  with DBDATE 3-23
  with DBFLTMASK 3-25
  with DBMONEY 3-27
  with DBTIME 3-32
  with GL_DATE 3-32
  with GL_DATETIME 3-32
  with USE_DTENV 3-32
FRACTION keyword
  DATETIME qualifier 2-12
FRAGMENT BY clause 3-30
Fragment-level statistics 1-29
Fragmentation
  distribution strategy 1-31
  encrypted distribution 1-29
  expression 1-29, 1-31, 3-22, 3-24
  fragment statistics 1-29
  list 1-31
  PDQPRIORITY environment variable 3-64
  PSORT_NPROCS environment variable 3-69
  round robin 1-29, 1-31
  setting priority levels for PDQ 3-64
  sysfragauth data 1-28
  sysfragdist data 1-29
  sysfragments data 1-31
FROM keyword 1-6, 1-15
Function keys 3-56
Functional index 1-33, 2-48, 3-36
Functions
  for BLOB columns 2-7
  for CLOB columns 2-11
  for MULTISET columns 2-25
  support for complex types 2-46
fwritable gcc option 3-50

G

gcc compiler 3-50
Generic B-trees 1-33
GET DIAGNOSTICS statement 1-26
getenv utility 3-2
GL_COLLATE table 1-52
GL_CTYPE table 1-52
GL_DATE environment variable 2-12, 3-22, 3-23
GL_DATETIME environment variable 2-12, 3-22
Global network buffer pool 3-44
GLS environment variables 3-5
GNU C compiler 3-50
GRANT statement 1-45, 1-51
Graphic characters 3-56
GROUP BY clause 2-8, 2-33, 3-30
GROUP BY TEXT 2-33
Group informix 3-26

H

Hash-join 3-61
hash() support function 2-46
Hashed columns 1-31
Hashing parameters 1-51
Heap size 3-58
Hebrew locales 2-9
Hexadecimal digits 3-24
HIGH INTEG keywords
  ALTER TABLE statement 2-40
  CREATE TABLE statement 2-40
HIGH keyword
  in UPDATE STATISTICS statement 1-29
  PDQPRIORITY 3-64
  UPDATE STATISTICS 1-6, 1-24
High-Performance Loader 3-28, 3-64
Histogram 1-24
Host language 1-65
Host variable 2-7, 2-8, 2-33, 2-48
HOUR keyword
  DATETIME qualifier 2-12
  INTERVAL qualifier 2-19
HP-UX operating system 3-70
HTML (Hypertext Markup Language) 2-11
Hyphen
  DATETIME delimiter 2-12
  INTERVAL delimiter 2-19

I

I/O overhead 3-69
IBM Informix ESQL/C 3-17, 3-23, 3-32, 3-43, 3-62
IDSSECURITYLABEL data type
  definition 2-18
IFMXMONGOAUTH environment variable 3-39
IFX_DEF_TABLE_LOCKMODE environment variable 3-40
IFX_DIRECTIVES environment variable 3-40
IFX_EXTDIRECTIVES environment variable 1-24, 3-41
IFX_LARGE_PAGES environment variable 3-42
IFX_LOB_XFERSIZE environment variable 3-43
IFX_LONGID environment variable 3-43
IFX_NETBUF_PVTPOOL_SIZE environment variable 3-44
IFX_NETBUF_SIZE environment variable 3-44
IFX_NO_SECURITY_CHECK environment variable 3-45
IFX_NO_TIMELIMIT_WARNING environment variable 3-45
IFX_NODBPROC environment variable 3-45
IFX_NOT_STRICT_THOUS_SEP environment variable 3-46
IFX_ONTAPE_FILE_PREFIX environment variable 3-46
IFX_PAD_VARCHAR environment variable 3-46
IFX_UNLOAD_EILSEQ_MODE environment variable 3-47
IFX_UPDDESC environment variable 3-47
IFX_XASTDCOMPLIANCE_XAEND environment variable 3-48
IFX_XFER_SHMBASE environment variable 3-48
imcadmin administrative tool 3-48
IMCADMIN environment variable 3-48
IMCCONFIG environment variable 3-49
IMCSERVER environment variable 3-49
IMPEXP data type 2-53
IMPEXPBIN data type 2-53
Implicit cast 1-14, 2-53
Implicit connection 3-54
Implicit temporary tables 3-30
import_binary() support function 2-46
import() support function 2-46
IN clause 3-30

IN keyword 2-8, 2-25, 2-29, 2-32, 2-55
IN TABLE storage option 3-36
Index
  attached 1-31, 3-22, 3-36, 3-69
  B-tree 1-33, 3-36
  clustered 1-33, 1-35
  composite 1-33
  default values for attached 3-69
  descending 1-33
  detached 3-36
  distribution scheme 3-36
  forest of trees 3-36
  fragmented 1-29, 1-31
  functional 1-33, 2-48, 3-36
  nonfragmented 3-36
  of data types 2-1
  of environment variables 3-9
  of system catalog tables 1-7
  R-Tree 3-36
  sysindexes data 1-33
  sysindices data 1-35
  sysobjstate data 1-38
  threads for sorting 3-69
  unique 1-22, 1-33, 2-30
Index key structure 1-35
Index privilege 1-51
Indirect typing 2-30
industry standards xiii
Industry standards, compliance with 1-65
INF_ROLE_SEP environment variable 3-57
Information Schema views
  accessing 1-63
  columns 1-64
  defined 1-62
  generating 1-63
  server_info 1-65
  sql_languages 1-65
  tables 1-64
Informational messages 1-26
Informix extension checking (DBANSIWARN) 3-19
informix owner name 1-6, 1-14, 1-24, 1-33, 1-35, 1-52, 3-26, 3-57
informix.rc file 3-2, 3-5, 3-40
INFORMIXC environment variable 3-50
INFORMIXCMCONUNITNAME environment variable 3-51
INFORMIXCMNAME environment variable 3-50
INFORMIXCONCSMCFG environment variable 3-51
INFORMIXCONRETRY environment variable 3-51
INFORMIXCONTIME environment variable 3-52
INFORMIXCPPMAP environment variable 3-54
INFORMIXDIR environment variable 3-54
INFORMIXSERVER environment variable 3-54
INFORMIXSHMBASE environment variable 3-55
INFORMIXSTACKSIZE environment variable 3-56
INFORMIXTERM environment variable 3-56
Inheritance hierarchy 1-37, 2-28
Initialization function 1-9, 1-45
Input support function 2-23
input() support function 2-46
Insert privilege 1-28, 1-51, 3-59
INSERT statements 1-55, 1-59, 2-12, 2-47, 3-18, 3-23
Insert trigger 1-57
Installation directory 3-54
INSTEAD OF trigger 1-57
INT data type 2-19
INT8 data type
  built-in casts 2-51, 2-52

INT8 data type *(continued)*
  coltype code 1-17
  defined 2-19
  using with SERIAL8 2-7
INTEG keyword 2-40
INTEGER data type
  built-in casts 2-51, 2-52
  coltype code 1-17
  defined 2-19
  length (syscolumns) 1-20
Intensity attributes 3-56
INTERACTIVE_DESKTOP_OFF environment variable 3-57
Internationalized trace messages 1-56
Interprocess communications (IPC) 3-55
INTERVAL data type
  coltype code 1-17
  defined 2-19
  field delimiters 2-19
  in expressions 2-41, 2-45, 2-46
  length (syscolumns) 1-20
ipcshm protocol 3-55
IS NULL operator 2-8
ISO 8859-1 code set 1-65
Isolation level 1-65, 3-61
Iterator functions 1-9

J

Japanese eras 3-32
Jar management procedures 3-58
JAR_TEMP_PATH environment variable 3-58
Java virtual machine (JVM) 3-16, 3-58
JAVA_COMPILER environment variable 3-58
JIT compiler 3-58
Join methods 3-61
Join operations 1-6, 3-30
JVM_MAX_HEAP_SIZE environment variable 3-58

K

KEEP ACCESS TIME keywords
  ALTER TABLE statement 2-40
  CREATE TABLE statement 2-40
Key
  primary 1-22, 1-44, 1-59, B-1
Key scan 1-10
Keyboard I/O
  INFORMIXTERM setting 3-56
  TERM setting 3-71
  TERMCAP setting 3-72
  TERMINFO setting 3-72
keyword MATCHES 2-33
Korn shell 3-2, 3-3

L

Label-based access control (LBAC) 2-18, 2-37
Language
  C 1-45, 3-17, 3-50
  C++ 3-54
  CLIENT_LOCALE setting 3-23
  DBLANG setting 3-26
  Extensible Markup Language (XML) 2-11
  Hypertext Markup Language (HTML) 2-11
  Informix ESQL/C 2-40, 2-48, 3-72
  Java 3-16, 3-58

Language *(continued)*
sql_languages information schema view 1-65
Stored Procedure Language (SPL) 2-48, 3-22, 3-24
syslangauth data 1-37
sysroutinelangs data 1-45
Large pages for virtual memory segments 3-42
Large-object data type
defined 2-39
listed 2-37
LD_LIBRARY_PATH environment variable 3-59
Leaf pages 1-31
libos.a library 3-43
LIBPATH environment variable 3-59
LIKE 2-33
LIKE keyword of SPL 2-30
LIKE operator 2-8, 2-55
Linearized code 1-56
List
of data types 2-1
of environment variables, by topic 3-9
of system catalog tables 1-7
LIST data type
coltype code 1-17
LIST data type, defined 2-22
LO_handles() support function 2-46
LOAD statement 2-7, 2-8, 2-33, 3-24
Locales
collation order 1-52
multibyte 2-10
of trace messages 1-56
right-to-left 2-9
specifying 3-9
Localized collation 2-37
Lock-table overflow 3-40
LOCKMODE keyword 3-40
LOCOPY function 2-7, 2-11
LOG keyword
ALTER TABLE statement 2-40
CREATE TABLE statement 2-40
Logging mode 1-16
Logical characters 2-37
Long identifiers
client version 3-43
IFX_LONGID setting 3-43
Information Schema views 1-63
LOTOFILE function 2-7, 2-11
LOW keyword
PDQPRIORITY 3-64
UPDATE STATISTICS 1-24
Lowercase mode codes 1-42
Lowercase privilege codes 1-1, 1-16, 1-51
LVARCHAR data type
casting opaque types 2-53
coltype code (for client) 1-17
defined 2-23

M

Machine notes 3-56
Machine-independent integer types 1-20
Magnetic storage media 1-14
Mantissa precision 1-64, 2-16
Map file for C++ programs 3-54
Mapping of data types
ANSI to Informix 2-6
MATCHES 2-33
MATCHES operator 2-8, 2-55
MaxConnect 3-48, 3-49
MEDIUM keyword 1-6, 1-21, 1-24
MEDIUM keyword, in UPDATE STATISTICS statement 1-29
Membership operator 2-55
MERGE statement 1-59
Message file
specifying subdirectory with DBLANG 3-26
Messages
chaining 3-62
error in syserrors 1-26
optimized transfers 3-62
reducing requests 3-62
trace message template 1-56
warning in syserrors 1-26
mi_collection_card() function 2-22, 2-25, 2-32
mi_db_error_raise() function 1-26
Microsoft C compiler 3-50
MINUTE keyword
DATETIME qualifier 2-12
INTERVAL qualifier 2-19
MITRACE_OFF configuration parameter 1-55, 1-56
mkdir utility 3-26
MODERATE INTEG keywords
ALTER TABLE statement 2-40
CREATE TABLE statement 2-40
Modifiers
CLASS 1-42
COSTFUNC 1-42
HANDLESNULLS 1-42
INTERNAL 1-42
NEGATOR 1-42
NOT VARIANT 1-42
PARALLELIZABLE 1-42
SELCONST 1-42
STACK 1-42
VARIANT 1-42
MODIFY NEXT SIZE keywords 1-6
MONEY data type
built-in casts 2-52
coltype code 1-17
defined 2-24
display format 3-27
international money formats 2-24
length (syscolumns) 1-20
MONTH keyword
DATETIME qualifier 2-12
INTERVAL qualifier 2-19
Multibyte characters
CLOB data type 2-11
MULTISET data type
coltype code 1-17
constructor 2-47
defined 2-25

N

N setting
sysroleauth.is_grantable 1-45
Named ROW data type
casting permitted 2-54
defined 2-27
defining 2-27
equivalence 2-27
inheritance 1-37, 2-27
typed tables 2-27
Namer ROW data type
coltype code 1-17

National Language Support (NLS) 2-37
NCHAR data type
  collation order 2-25
  coltype code 1-17
  defined 2-25
  multibyte characters 2-25
Negator functions 1-42
Nested dot notation 2-48
Nested-loop join 3-61
Network buffers 3-44
Network environment variable, DBPATH 3-28
NFS directory 3-32
NLS data types
  in system catalog tables 1-7
NO KEEP ACCESS TIME keywords
  ALTER TABLE statement 2-40
  CREATE TABLE statement 2-40
no setting of NODEFDAC 3-59
NODEFDAC environment variable 3-59
NOLOG keyword
  ALTER TABLE statement 2-40
  CREATE TABLE statement 2-40
Non-default database locales 1-7
NONE setting
  JAVA_COMPILER 3-58
Nonfragmented index 3-36
Nonprintable characters
  CHAR data type 2-10
  TEXT data type 2-35
  VARCHAR data type 2-35
NOT NULL 2-33
NOT NULL constraint
  collection elements 2-22, 2-25, 2-32, 2-47
  syscoldepend data 1-17
  sysconstraints data 1-22
NOT NULL keywords 2-8, 2-22
NOT operator 2-55
NOT VARIANT routine 1-42
NULL data type
  coltype code 1-17
NULL value
  allowed or not allowed 1-9, 1-17
  BOOLEAN literal 2-8
  BYTE data type 2-8
Numeric data types
  casting between 2-51
  casting to character types 2-52
  listed 2-37
NVARCHAR data type
  collation order 2-26
  coltype code 1-17
  defined 2-26
  multibyte characters 2-26

O

Object mode of database objects 1-38
Object-relational schema B-1
ODBC driver 3-59, 3-70
OFF setting
  IFX_DIRECTIVES 3-40, 3-41
  PDQPRIORITY 3-64
ON setting
  IFX_DIRECTIVES 3-40, 3-41
ONCONFIG environment variable 3-60
onconfig.std file 3-71
oninit command 3-40
ONINIT_STDOUT environment variable 3-60
Online transaction processing (OLTP) 1-31
onload utility 2-7, 2-8, 2-33
onpload utility 3-28, 3-65
onsecurity utility 3-45
onstat utility 3-1
Opaque data types
  cast matrix 2-54
  comparing 2-53
  storage 2-23
  sysxtddesc data 1-60
  sysxtdtypes data 1-61
OPAQUE data types
  defined 2-26
OPEN statement 3-62
Operator class
  sysams data 1-10
  sysindices data 1-35
  sysopclasses data 1-39
operator LIKE 2-33
Operator precedence 2-55
operator TEXT 2-33
OPT_GOAL configuration parameter 3-63
OPT_GOAL environment variable 3-63
OPTCOMPIND configuration parameter 3-61
OPTCOMPIND environment variable 3-61
Optimizer
  setting IFX_DIRECTIVES 3-40
  setting IFX_EXTDIRECTIVES 3-41
  setting OPT_GOAL 3-63
  setting OPTCOMPIND 3-61
  setting OPTOFC 3-62
Optimizer directives
  sysdirectives data 1-24
OPTMSG environment variable 3-62
OPTOFC environment variable 3-62
OR operator 2-55
ORDER 2-33
ORDER BY clause 2-8, 3-30
Ordinal positions 2-22
Output support function 2-23
output() support function 2-46
Overflow error 2-16
Owner routines 1-42, 3-59

P

Page footers in sbspaces 2-40
Page headers in sbspaces 2-40
PAGE lock mode 1-52, 3-40
Parallel distributed queries, setting with PDQPRIORITY 3-64
Parallel sorting, setting with PSORT_NPROCS 3-68
Partial characters 2-9
Partial-column index 1-35
PATH environment variable 3-63
Pathname
  Configuration file
    for terminal I/O 3-72
  for C compiler 3-50
  for C++ map file 3-54
  for concsm.cfg file 3-51
  for connectivity information 3-55
  for database server 3-28
  for dynamic-link libraries 3-59, 3-70
  for environment-configuration file 3-4
  for executable programs 3-63
  for installation 3-54

Pathname *(continued)*
  for message files 3-26
  for parallel sorting 3-68
  for remote shell 3-30
  for smart-large-object handles 3-65
  for temporary .jar files 3-58
  for termcap file 3-72
  for terminfo directory 3-72
  separator symbols 3-63
PDQ
  OPTCOMPIND environment variable 3-61
  PDQPRIORITY environment variable 3-64
Percentage (%) symbol 3-32
Period
  DATE delimiter 3-23
  DATETIME delimiter 2-12
  INTERVAL delimiter 2-19
Permissions 3-2, 3-26
PLCONFIG environment variable 3-64
plconfig file 3-64
PLOAD_LO_PATH environment variable 3-65
PLOAD_SHMBASE environment variable 3-65
PostScript 2-11
Precedence rules
  for casts 2-53
  for lock mode 3-40
  for SQL operators 2-55
  for UNIX environment variables 3-5
  for Windows environment variables 3-8
Precision
  of currency values 2-24
  of numbers 1-64, 2-15, 2-18, 2-19, 2-33
  of time values 2-12, 2-19, 2-42, 2-45
PREPARE statement 1-52
Prepared statement 1-52
Primary access method 1-10, 1-51
Primary key 1-22, 1-44, 1-59, 2-30, B-1
Primary thread 3-56
printenv utility 3-4
Printing with DBPRINT 3-30
Private environment-configuration file 3-4, 3-38
Private network buffer pool 3-44
Private synonym 1-52
Privilege
  default table privileges 3-59
  on columns (syscolauth table) 1-16
  on procedures and functions (sysprocauth table) 1-40
  on table fragments (sysfragauth table) 1-28
  on tables (systabauth table) 1-51
  on the database (sysusers table) 1-58
  on UDTs and named row types (sysxtdtypeauth) 1-60
Protected routines 1-42
Protected rows 2-18, 2-37
Pseudo-machine code (p-code) 1-40
PSM_ACT_LOG environment variable 3-66
PSM_CATALOG_PATH environment variable 3-66
PSM_DBS_POOL environment variable 3-66
PSM_DEBUG environment variable 3-67
PSM_DEBUG_LOG environment variable 3-67
PSM_LOG_POOL environment variable 3-68
PSORT_DBTEMP environment variable 3-68
PSORT_NPROCS environment variable 3-68
Public synonym 1-50, 1-52
public user name 1-63
Purpose functions 1-10
putenv utility 3-2

Q

Qualifier field
  DATETIME 2-12
  EXTEND 2-45
  INTERVAL 2-19
  UNITS 2-44
Query optimizer
  directives 3-40, 3-41
  sysdistrib data 1-24
  sysprocplan data 1-44
  updating distribution data 1-6
Quoted string
  DATE and DATETIME literals 2-44
  DELIMIDENT setting 3-37
  INTERVAL literals 2-19
  invalid with BYTE 2-8
  LVARCHAR data type 2-23
Quoted string invalid with TEXT 2-33

R

R-tree index 3-36, 3-69
Read committed 1-65
Read uncommitted 1-65
recv() support function 2-46
References privilege 1-16, 1-51
Referential constraint 1-22, 1-44, 1-59
Relational operators 2-10, 2-55
Remote database server 1-50, 3-39
Remote shell 3-30
Remote tape devices 3-30
RENAME SEQUENCE statement 3-73
Repeatable read 3-61
Replica identifier 1-31
RESIDENT configuration parameter 3-42
Resource contention 3-64
Resource Grant Manager (RGM) 1-31
Resource privilege 1-6
  Role
    sysusers data 1-58
  System catalog
    authorization identifiers 1-58
REVOKE statement 1-51
Right-to-left locales 2-9
Role
  default role 1-58
  INF_ROLE_SEP setting 3-57
  sysroleauth data 1-45
Role separation 3-57
Rolling-window fragmentation 1-31
Round-robin fragmentation 1-29, 1-31
Routines
  DataBlade API routine 1-55
  DATETIME formatting 3-32
  identifier 1-42
  owner 1-42
  privileges 1-40
  protected 1-42
  restricted 1-42
  Stored Procedure Language (SPL) 2-48
  syserrors data 1-26
  syslangauth data 1-37
  sysprocauth data 1-40
  sysprocbody data 1-40
  sysprocedures data 1-42
  sysprocplan data 1-44

Routines *(continued)*
  sysroutinelangs data 1-45
  systraceclasses data 1-55
  systracemsgs data 1-56
  trigger 1-42
ROW data types 2-48
  casting permitted 2-54
  equivalence 2-27
  fields 1-12, 2-48
  inheritance 1-37, 2-27
  inserting values 2-29
  named 2-27, 2-48
  sysattrtypes data 1-12
  sysxtddesc data 1-60
  sysxtdtypes data 1-60, 1-61
  unnamed 2-28, 2-48
ROW lock mode 1-52, 3-40
ROWIDS 1-10
RTNPARAMTYPES data type 1-42
RTREE_COST_ADJUST_VALUE environment variable 3-69
Runtime
  warnings (DBANSIWARN) 3-19

S

Sample size 1-24
Sampling data 1-29
SAVE EXTERNAL DIRECTIVES statement 3-41
SBSPACENAME configuration parameter 1-24, 1-29
sbspaces
  defined 2-11, 2-40
  name 3-37
  sysams data 1-10
  syscolattribs data 1-16
  systabamdata data 1-51
Scale of numbers 1-64, 2-16, 3-25
Scan cost 1-10
Schema Tools 3-6
Screen reader
  reading syntax diagrams C-1
SECOND keyword
  DATETIME qualifier 2-12
  FRACTION keyword
    INTERVAL qualifier 2-19
  INTERVAL qualifier 2-19
Secondary-access methods 1-10, 1-21, 1-35, 1-39, 2-26
Security policy 2-18
SELECT INTO TEMP statement 3-30
Select privilege 1-16, 1-51, 1-63, 3-59
SELECT statements 1-6, 1-24
SELECT triggers 1-57
Selectivity constant 1-42
Self-join 1-1
send() support function 2-46
SENDRECV data type 2-53
Sequence
  syssequences data 1-49
  syssynonyms data 1-50
  syssyntable data 1-50
  systabauth data 1-51
  systables data 1-52
Sequential integers
  am_id code 1-10
  classid code 1-55
  constrid code 1-22
  extended_id code 1-61
  langid code 1-45
Sequential integers *(continued)*
  msgid code 1-56
  opclassid code 1-39
  planid code 1-44
  procid code 1-42
  seqid code 1-49
  SERIAL data type 2-30
  SERIAL8 data type 2-30
  tabid code 1-1, 1-49, 1-52
SERIAL data type
  coltype code 1-17
  defined 2-30
  inserting values 2-30
  length (syscolumns) 1-20
  resetting values 2-30
SERIAL8 data type
  assigning a starting value 2-31
  coltype code 1-17
  defined 2-30
  inserting values 2-31
  length (syscolumns) 1-20
  resetting values 2-31
  using with INT8 2-7
Serializable transactions 1-65
server_info Information Schema view 1-62
SET data type
  coltype code 1-17
SET data type, defined 2-32
SET ENVIRONMENT IFX_AUTO_REPREPARE statement 1-52
SET ENVIRONMENT statement 3-2, 3-6, 3-61
SET OPTIMIZATION statement 3-63
SET PDQPRIORITY statement 3-64
SET SESSION AUTHORIZATION statement 1-42
SET STMT_CACHE statement 3-71
set utility 3-7
setenv utility 3-4
Setnet32 3-8
Setnet32 utility 3-6
Setting environment variables
  in UNIX 3-2
  in Windows 3-6
SGML (Standard Graphic Markup Language) 2-11
Shared environment-configuration file 3-4
Shared libraries 3-43
Shared memory
  INFORMIXSHMBASE 3-55
  PLOAD_SHMBASE 3-65
Shell
  remote 3-30
  search path 3-63
  setting environment variables in a file 3-3
  specifying with DBREMOTECMD 3-30
SHLIB_PATH environment variable 3-70
Shortcut keys
  keyboard C-1
simple large object
  defined 2-8
Simple large objects
  defined 2-40
  location (sysblobs) 1-14
Single-precision floating-point number 2-27, 2-33
SMALLFLOAT data type
  built-in casts 2-51, 2-52
  coltype code 1-17
  defined 2-33
  display format 3-25, 3-27

SMALLINT data type
  built-in casts 2-51, 2-52
  coltype code 1-17
  defined 2-33
  length (syscolumns) 1-20
Smart large objects
  defined 2-40
  syscolattribs data 1-16
Smart-large-object handles 3-65
Solaris operating system 3-42
SOME operator 2-55
Sort-merge join 3-61
Sorting
  DBSPACETEMP environment variable 3-30
  PSORT_DBTEMP environment variable 3-68
  PSORT_NPROCS environment variable 3-68
Space
  DATETIME delimiter 2-12
  INTERVAL delimiter 2-19
Spatial queries 3-69
SPL routines 1-42, 2-48, 3-22, 3-24
SPL variables 2-48
SQL (Structured Query Language) 3-19
SQL character set 3-37
SQL Communications Area 3-19
sql_languages Information Schema view 1-62
SQL_LOGICAL_CHAR configuration parameter 1-52, 2-37
sqlhosts file 3-48, 3-54, 3-55
SQLHOSTS subkey 3-55
SQLSTATE values 1-26
sqltypes.h file 1-17
SQLWARN array 3-19
SRV_FET_BUF_SIZE environment variable 3-70
Stack size 1-42, 3-56
STACKSIZE configuration parameter 3-56
Standard Graphic Markup Language (SGML) 2-11
standards xiii
START DATABASE statement 3-28
START VIOLATIONS TABLE statement 1-59
STAT data type 1-24
STATCHANGE configuration parameter 1-24, 1-29
STATCHANGE table attribute 1-24, 1-29
Statement cache 3-71
Statements of SQL
  ALTER INDEX 1-35
  ALTER SEQUENCE 1-49, 3-73
  ALTER TABLE 1-6, 1-44, 1-52, 3-73
  CLOSE 3-62
  CONNECT 3-28, 3-52, 3-54
  CREATE ACCESS_METHOD 1-10
  CREATE AGGREGATE 1-9
  CREATE CAST 1-14, 2-52
  CREATE DATABASE 3-28
  CREATE DISTINCT TYPE 1-61, 2-17, B-2
  CREATE EXTERNAL TABLE 1-27, 1-28
  CREATE FUNCTION 1-45, 3-59
  CREATE IMPLICIT CAST B-2
  CREATE INDEX 1-1, 1-33, 1-35, 1-52, 3-36
  CREATE OPAQUE TYPE 1-61, 2-26
  CREATE OPERATOR CLASS 1-39
  CREATE PROCEDURE 1-40, 1-45
  CREATE ROLE 1-45, 1-58
  CREATE ROUTINE FROM 1-45
  CREATE ROW TYPE 1-61, 2-27
  CREATE SCHEMA AUTHORIZATION 1-1
  CREATE SEQUENCE 1-49
  CREATE SYNONYM 1-50
Statements of SQL *(continued)*
  CREATE TABLE 1-22, 1-44, 1-51
  CREATE TRIGGER 1-57
  CREATE VIEW 1-58
  CREATE XADATASOURCE 1-59
  CREATE XADATASOURCETYPE 1-60
  DATABASE 3-28
  DECLARE 3-62
  DELETE 1-6, 1-44, 1-59
  DESCRIBE 3-47
  DROP CAST B-2
  DROP DATABASE 3-28
  DROP FUNCTION 1-42
  DROP INDEX 1-52
  DROP PROCEDURE 1-42
  DROP ROUTINE 1-42
  DROP ROW TYPE 2-27
  DROP SEQUENCE 3-73
  DROP TABLE 3-73
  DROP TYPE 2-17, 2-26
  DROP VIEW 1-63, 3-73
  FETCH 3-62
  GET DIAGNOSTICS 1-26
  GRANT 1-28, 1-45, 1-51, 1-63
  INSERT 1-59, 2-47, 3-18, 3-23
  LOAD 2-8, 3-19
  MERGE 1-59
  OPEN 3-62
  PREPARE 1-52
  RENAME SEQUENCE 3-73
  RENAME TABLE 3-73
  REVOKE 1-51, 1-58
  SELECT 1-6, 1-24, 1-44, 3-30
  SET ENVIRONMENT 3-61
  SET OPTIMIZATION 3-63
  SET PDQPRIORITY 3-64
  SET SESSION AUTHORIZATION 1-42
  SET STMT_CACHE 3-71
  START DATABASE 3-28
  START VIOLATIONS TABLE 1-59
  UNLOAD 3-20
  UPDATE 3-18
  UPDATE STATISTICS 1-6, 1-35, 3-35
  UPDATE STATISTICS FOR PROCEDURE 1-44
  UPDATE STATISTICS FOR TABLE 1-21
Statements of SQL LOAD 2-33
Statements of SQL UPDATE 2-33
static option of ESQL/C 3-43
STATLEVEL table attribute 1-29
STMT_CACHE configuration parameter 3-71
STMT_CACHE environment variable 3-71
STMT_CACHE keyword 3-71
Storage identifiers 3-37
Stored procedure language (SPL) 1-42, 2-48, 3-22
stores_demo database A-1
  join columns A-1
strings option of gcc 3-50
Structured Query Language (SQL) 3-19
Subscripts 2-8
Subscripts ([ ]), 2-33
SUBSTRING function 1-6
Subtable 1-29, 1-31, 1-37, B-3
Subtype 1-37, 2-27
Summary
  of data types 2-1
  of environment variables, by topic 3-9
  of system catalog tables 1-7

superstores_demo database
  structure of tables B-1
Supertable 1-37, B-3
Supertype 1-37, 2-27
Support functions
  DISTINCT data types 2-49
  OPAQUE data types 2-26, 2-46
  routine identifier 1-42
Symbol table 1-42
Synonym
  syssynonyms data 1-50
  syssyntable data 1-50
  systables data 1-52
  USETABLENAME setting 3-73
Syntax diagrams
  reading xiii
  reading in a screen reader C-1
sysaggregates system catalog table 1-9
sysams system catalog table 1-10
sysattrtypes system catalog table 1-12
sysautolocate system catalog table 1-13
sysblobs system catalog table 1-14
sysbuiltintypes table 1-1
syscasts system catalog table 1-14, 2-50
syschecks system catalog table 1-15
syscheckudrdep system catalog table 1-15
syscolattribs system catalog table 1-16
syscolauth system catalog table 1-16
syscoldepend system catalog table 1-17
syscolumns system catalog table 1-17
sysconstraints system catalog table 1-22
syscrd database 1-1
sysdbclose
  disabling with IFX_NODBPROC 3-45
sysdbopen
  disabling with IFX_NODBPROC 3-45
sysdefaults system catalog table 1-22
sysdepend system catalog table 1-23
sysdirectives system catalog table 1-24
sysdistrib system catalog table 1-24
sysdomains system catalog view 1-26
syserrors system catalog table 1-26
sysextcols system catalog table 1-27
sysextdfiles system catalog table 1-27
sysexternal system catalog table 1-28
sysfragauth system catalog table 1-28
sysfragdist system catalog table 1-29
sysfragments system catalog table 1-31
sysindexes system catalog table 1-33
sysindexes system catalog tables 1-35
sysinherits system catalog table 1-37
syslangauth system catalog table 1-37
syslogmap system catalog table 1-38
sysmaster database 1-1
  contrasted with system catalog tables 1-1
  initialization 3-1
sysobjstate system catalog table 1-38
sysopclasses system catalog table 1-39
sysprocauth system catalog table 1-40
sysprocbody system catalog table 1-40
sysproccolumns system catalog table 1-41
sysprocedures system catalog table 1-42
sysprocplan system catalog table 1-44
sysreferences system catalog table 1-44
sysroleauth system catalog table 1-45
sysroutinelangs system catalog table 1-45
sysseclabelauth system catalog table 1-46
sysseclabelcomponentelements system catalog table 1-46
sysseclabelcomponents system catalog table 1-46
sysseclabelnames system catalog table 1-47
sysseclabels system catalog table 1-47
syssecpolicies system catalog table 1-47
syssecpolicycomponents system catalog table 1-48
syssecpolicyexemptions system catalog table 1-48
syssequences system catalog table 1-49
syssurrogateauth system catalog table 1-49
syssynonyms system catalog table 1-50
syssyntable system catalog table 1-50
systabamdata system catalog table 1-51
systabauth system catalog table 1-51
systables system catalog table 1-52
System administrator (DBA) 1-1
System applet 3-6
System catalog
  access methods 1-10, 1-51
  access privileges 1-16, 1-28
  accessing 1-6
  altering contents 1-6
  casts 1-14
  columns 1-17
  complex data types 1-12, 1-61
  constraint violations 1-59
  constraints 1-15, 1-17, 1-22
  data distributions 1-24
  database tables 1-52
  default values 1-22
  defined 1-1
  dependencies 1-23
  discretionary access privileges 1-51
  drvurity policies 1-47
  example 1-1
  external directives 1-24
  external tables 1-27, 1-28
  fragment distributions 1-29
  fragment privileges 1-28
  fragments 1-31
  indexes 1-33, 1-35
  inheritance 1-37
  list of tables 1-7
  messages 1-26
  operator classes 1-39
  privileges 1-58, 1-60
  programming languages 1-37, 1-45
  referential constraints 1-22, 1-44, 1-59
  roles 1-45
  routine parameters 1-41
  routines 1-40, 1-42, 1-44
  security label components 1-46
  sequence objects 1-49
  simple large objects 1-14
  smart large objects 1-16
  synonyms 1-50
  text of routines 1-40
  trace classes 1-55
  trace messages 1-56
  triggers 1-56, 1-57
  updating 1-6
  use by database server 1-1
  user-defined aggregates 1-9
  user-defined data types 1-60, 1-61
  views 1-52, 1-58
  XA data source types 1-60
  XA data sources 1-59

System catalog tables
  synonyms 1-50
  sysaggregates 1-9
  sysams 1-10
  sysattrtypes 1-12
  sysautolocate 1-13
  sysblobs 1-14
  syscasts 1-14
  syschecks 1-15
  syscheckudrdep 1-15
  syscolattribs 1-16
  syscolauth 1-16
  syscoldepend 1-17
  syscolumns 1-17
  sysconstraints 1-22
  sysdefaults 1-22
  sysdepend 1-23
  sysdirectives 1-24
  sysdistrib 1-24
  sysdomains 1-26
  syserrors 1-26
  sysextcols 1-27
  sysextdfiles 1-27
  sysexternal 1-28
  sysfragauth 1-28
  sysfragdist 1-29
  sysfragments 1-31
  sysindexes 1-33
  sysindices 1-35
  sysinherits 1-37
  syslangauth 1-37
  syslogmap 1-38
  sysobjstate 1-38
  sysopclasses 1-39
  sysprocauth 1-40
  sysprocbody 1-40
  sysproccolumns 1-41
  sysprocedures 1-42
  sysprocplan 1-44
  sysreferences 1-44
  sysroleauth 1-45
  sysroutinelangs 1-45
  sysseclabelauth 1-46
  sysseclabelcomponentelements 1-46
  sysseclabelcomponents 1-46
  sysseclabelnames 1-47
  sysseclabels 1-47
  syssecpolicies 1-47
  syssecpolicycomponents 1-48
  syssecpolicyexemptions 1-48
  syssequences 1-49
  syssurrogateauth 1-49
  syssynonyms 1-50
  syssyntable 1-50
  systabamdata 1-51
  systabauth 1-51
  systables 1-52
  systraceclasses 1-55
  systracemsgs 1-56
  systrigbody 1-56
  systriggers 1-57
  sysusers 1-58
  sysviews 1-58
  sysviolations 1-59
  sysxadatasources 1-59
  sysxasourcetypes 1-60
  sysxtddesc 1-60
System catalog tables *(continued)*
  sysxtdtypeauth 1-60
  sysxtdtypes 1-61
SYSTEM() command, on NT 3-57
systraceclasses system catalog table 1-55
systracemsgs system catalog table 1-56
systrigbody system catalog table 1-56
systriggers system catalog table 1-57
sysusers system catalog table 1-58
sysutils database 1-1
sysuuid database 1-1
sysviews system catalog table 1-58
sysviolations system catalog table 1-59
sysxadatasources system catalog table 1-59
sysxasourcetypes system catalog table 1-60
sysxtddesc system catalog table 1-60
sysxtdtypeauth system catalog table 1-60
sysxtdtypes system catalog table 1-61, 2-26, 2-27

T

tabid 1-1, 1-52
Table
  changing a column data type 2-50
  dependencies, in sysdepend 1-23
  diagnostic 1-59
  extent size 1-52
  fragmented 1-29, 1-31
  hashing parameters 1-51
  hierarchy 1-29, 1-31, 1-37, 2-27, B-3
  inheritance, sysinherits data 1-37
  lock mode 1-52, 3-40
  nonfragmented 3-36
  separate from large object storage 2-39
  structure in superstores_demo database B-1
  synonyms in syssyntable 1-50
  systables data 1-52
  system catalog tables 1-9
  temporary 3-30, 3-32
  temporary in SE 3-32
  untyped, and unnamed ROW 2-29
  version value 1-52
  violations 1-59
Table-based fragmentation 1-31
Table-level privileges
  PUBLIC 1-63
  sysfragauth data 1-28
  systabauth data 1-1, 1-51
tables Information Schema view 1-62
Tape management
  setting DBREMOTECMD 3-30
Temporary dbspace 3-30
Temporary files 3-32
  in SE, specifying directory with DBTEMP 3-32
  setting DBSPACETEMP 3-30
  setting PSORT_DBTEMP 3-68
Temporary tables 3-30
  in SE, specifying directory with DBTEMP 3-32
  specifying dbspace with DBSPACETEMP 3-30
TERM environment variable 3-71
TERMCAP environment variable 3-72
termcap file
  setting INFORMIXTERM 3-56
  setting TERMCAP 3-72
Terminal handling
  setting INFORMIXTERM 3-56
  setting TERM 3-71

Terminal handling *(continued)*
  setting TERMCAP 3-72
  setting TERMINFO 3-72
terminfo directory 3-56, 3-72
TERMINFO environment variable 3-72
TEXT 2-33
TEXT argument 2-33
TEXT Character string TEXT 2-33
TEXT data type 2-33
  coltype code 1-17
  increasing buffer size 3-20
  length (syscolumns) 1-20
  nonprintable characters 2-35
  setting buffer size 3-20
  sysblobs data 1-14
  sysfragments data 1-31
  with control characters 2-35
TEXT data type IS NULL 2-33
TEXT data type restrictions 2-33
Text editor 3-25
thousands separator 3-46
Thousands separator 2-24
thread flag of ESQL/C 3-72
THREADLIB environment variable 3-72
Time data types
  arithmetic 2-41
  length (syscolumns) 1-20
  listed 2-37
Time values
  DBCENTURY setting 3-20
  DBDATE setting 3-23
  DBTIME setting 3-32
  GL_DATETIME settings 3-32
  USEOSTIME configuration parameter 2-12
Time-limited licenses (IFX_NO_TIMELIMIT_WARNING) 3-45
Timezone
  setting TZ 3-73
TO keyword
  DATETIME qualifier 2-12
  EXTEND function 2-43
  INTERVAL qualifier 2-19
TODAY operator 1-22
Trace class 1-55
Trace messages 1-56
Trace statements 1-56
Transaction isolation level 1-65, 3-61
Transaction logging 1-16, 1-65
Trigger routines 1-42
Triggers
  creation-time value 3-22, 3-24
  sysobjstate data 1-38
  systrigbody data 1-56
  systriggers data 1-57
TRUE setting
  BOOLEAN values 2-8
  sysams table 1-10
Truncation 2-9
TYPE keyword 2-28
TZ environment variable 3-73

U

UDT indexes 3-69
Unary arithmetic operators 2-55
Uncommitted read 1-65
Under privilege 1-51
Unique constraint 1-59, 2-30
Unique index 1-33, 2-30
Unique keys 1-10
Unique numeric values
  SERIAL data type 2-30
  SERIAL8 data type 2-30
UNITS operator 2-12, 2-41, 2-44, 2-55
UNIX
  BSD, default print utility 3-30
  environment variables 3-1
  PATH environment variable 3-63
  System V
    default print utility 3-30
    terminfo libraries 3-56, 3-72
  temporary files 3-68
  TERM environment variable 3-71
  TERMCAP environment variable 3-72
  TERMINFO environment variable 3-72
UNLOAD statement 3-20, 3-24
Unnamed ROW data type
  coltype code 1-17
  declaring 2-29
  defined 2-28
  inserting values 2-29
unset utility 3-4
unsetenv utility 3-4
Unsetting an environment variable 3-4
Untyped table 1-52
Update privilege 1-16, 1-28, 1-51, 3-59
UPDATE statement 1-59
UPDATE statements 3-47
UPDATE STATISTICS FOR PROCEDURE statement 1-44
UPDATE STATISTICS statement 1-35, 3-35
  and DBUPSPACE environment variable 3-35
  effect on sysdistrib table 1-24
  sysindices data 1-35
  updating system catalog tables 1-6
Update trigger 1-57
Uppercase mode codes 1-42
Uppercase privilege codes 1-1, 1-16, 1-51
USE_DTENV environment variable 2-12
USEOSTIME configuration parameter 2-12
User environment variable 3-8
User informix 1-6, 1-14, 2-50
User name 1-65
User privileges
  syscolauth data 1-16
  sysfragauth data 1-28
  syslangauth data 1-37
  sysprocauth data 1-40
  systabauth data 1-51
  sysusers data 1-58
  sysxtdtypeauth data 1-60
User-defined aggregates 1-9
User-defined casts 2-52
User-defined casts (UDCs) 1-14
User-defined data types
  casting 2-52
  casting into built-in type 2-50
  opaque 2-49
  sysxtddesc data 1-60
  sysxtdtypes data 1-60, 1-61
User-defined routines
  casts (syscasts) 1-14
  check constraints (syscheckudrdep) 1-15
  error messages (syserrors) 1-26
  for OPAQUE data types 2-26
  functional index 3-36

User-defined routines *(continued)*
  language authorization (syslangauth) 1-37
  privileges 1-40, 3-59
  protected 1-42
  secondary access method 1-21
  sysprocedures data 1-42
USETABLENAME environment variable 3-73
Utilities
  chkenv 3-2, 3-4
  DB-Access 1-6, 1-63, 3-6, 3-19, 3-25, 3-54
  dbload 2-7, 2-8
  dbschema 1-42
  env 3-4
  export 3-3
  gcc 3-50
  getenv 3-2
  ifx_getenv 3-6
  ifx_putenv 3-6
  imcadmin 3-48
  lp 3-30
  lpr 3-30
  MaxConnect 3-49
  oninit 3-40
  onload 2-7, 2-8
  onpload 3-28, 3-65
  onsecurity 3-45
  printenv 3-4
  putenv 3-2
  set 3-7
  setenv 3-4
  Setnet32 3-6
  source 3-2
  unset 3-4
  unsetenv 3-4, 3-37
  vi 3-25
Utilities dbload 2-33

V

VARCHAR data type
  ([ ]), brackets
    MATCHES range delimiters 2-35
  CHAR data type
    collation 2-35
  Code sets
    collation order 2-35
    East Asian 2-35
  Collation
    VARCHAR data type 2-35
  coltype code 1-17
  defined 2-35
  Locales
    collation order 2-35
  MATCHES operator 2-35
  Multibyte characters
    VARCHAR data type 2-35
  nonprintable characters 2-35
  SQL_LOGICAL_CHAR configuration parameter 2-35
  storing numeric values 2-35
  VARCHAR data type
    collation 2-35
    multibyte characters 2-35
  Zero (0)
    C null as terminator 2-35
Variable-length opaque data types 1-17
Variable-length packets 3-46
Variable-length UDT 1-61
VARIANT routine 1-42
Version of a table 1-52
vi text editor 3-25
View
  columns view 1-64
  Information Schema 1-62
  server_info view 1-65
  sql_languages view 1-65
  sysdepend data 1-23
  sysindexes view 1-35
  syssynonyms data 1-50
  syssyntable data 1-50
  systabauth data 1-51
  systables data 1-52
  sysviews data 1-58
  tables view 1-64
Violations
  sysobjstate data 1-38
  sysviolations data 1-59
Virtual machine 3-16, 3-58
Virtual processors 3-69
Visual disabilities
  reading syntax diagrams C-1

W

Warning message 1-26, 3-19
WHERE 2-33
WHERE keyword 1-6, 1-15
Whitespace in identifiers 3-37
Window borders 3-56
Windows environments
  manipulating environment variables 3-6
  setting environment variables 3-6

X

X setting
  sysams.am_sptype 1-10
  systabauth.tabauth 1-51
X/Open
  compliance 1-65
  server_info view 1-65
X/Open CAE standards 1-62
XA data source types 1-60
XA data sources 1-59
XML (Extensible Markup Language) 2-11
XPG4 standard 1-64

Y

Y setting
  DBDATE 3-23
  DBTIME 3-32
  sysroleauth.is_grantable 1-45
Year 2000 3-20
YEAR keyword
  DATETIME qualifier 2-12
  EXTEND function 2-43
  INTERVAL qualifier 2-19
Year values, two and four digit 2-12, 3-20, 3-23, 3-32
yes setting
  NODEFDAC 3-59
YES setting
  columns.is_nullable 1-64
  sql_languages.integrity 1-65

Z

Zero
- extent size encoding 1-35

Zero (0)
- DBDATE separator 3-23
- DECIMAL scale 2-15
- hexadecimal digit 3-24
- IFX_DIRECTIVES setting 3-40, 3-41
- IFX_LARGE_PAGES setting 3-42
- IFX_LONGID setting 3-43
- IFX_NETBUF_PVTPOOL_SIZE setting 3-44
- integer scale 1-64, 2-15
- OPTCOMPIND setting 3-61
- OPTMSG setting 3-62
- padding of 1-digit years 3-20
- padding with DBFLTMASK 3-25
- padding with DBTIME 3-32
- PDQPRIORITY setting 3-64
- PSORT_NPROCS setting 3-69
- STMT_CACHE setting 3-71
- sysams values 1-10
- sysfragments.hybdpos 1-31
- sysindices.nrows 1-35
- systables.type_xid 1-52
- sysxdtypes values 1-61

Printed in USA

SC27-4522-05

Spine information:

Informix Product Family Informix **Version 12.10** **IBM Informix Guide to SQL: Reference**