IBM Informix
Version 11.50

IBM Informix Guide to SQL: Reference

SC27-3621-01

IBM Informix
Version 11.50

IBM Informix Guide to SQL: Reference

SC27-3621-01

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

Edition

This edition replaces SC27-3621-00.

Contents

Introduction

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

If you have limited experience with relational databases, SQL, or your operating system, see the *IBM Informix Getting Started Guide* for your database server for a list of supplementary titles.

Software Dependencies

This publication is written with the assumption that you are using one of the following database servers:

- IBM Informix, Version 11.50
- IBM Informix Extended Parallel Server, Version 8.51

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 is brought together in a single environment, called a Global Language Support (GLS) locale.

This publication assumes that your database uses the default locale. This default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) in Windows environments. This locale supports U.S. English format conventions for displaying and entering date, time, number, and currency values. It also supports the ISO 8859-1 (on UNIX and Linux) or Microsoft 1252 (on Windows) code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or in SQL identifiers, or if you plan to use other collation rules for sorting character data, you must specify the appropriate nondefault locale.

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

Demonstration Database

The DB–Access utility, which is provided with the 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.
- **Extended Parallel Server:** The **sales_demo** database illustrates a dimensional schema for data- warehousing applications. For conceptual information about dimensional data modeling, see the *IBM Informix Database Design and Implementation Guide*.
- **IBM Informix:** 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 Appendix A, "The stores_demo Database," on page A-1 and Appendix B, "The sales_demo and superstores_demo Databases," on page B-1.

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

What's New in SQL Reference for IBM Informix, Version 11.50

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

The following changes and enhancements are relevant to this publication. For a comprehensive list of all new features for this release, see the *IBM Informix Getting Started Guide*.

The following table lists the new features for Version 11.50.xC8.

Table 1. What's New in the IBM Informix Guide to SQL: Reference for Version 11.50.xC8

Overview	Reference
New editions and product names IBM Informix Dynamic Server editions were withdrawn and new Informix editions are available. Some products were also renamed. The publications in the Informix library pertain to the following products: • IBM Informix database server, formerly known as IBM Informix Dynamic Server (IDS) • IBM OpenAdmin Tool (OAT) for Informix, formerly known as OpenAdmin Tool for Informix Dynamic Server (IDS) • IBM Informix SQL Warehousing Tool, formerly known as Informix Warehouse Feature	For more information about the Informix product family, go to http://www.ibm.com/software/data/informix/.

Table 2. What's New in IBM Informix Guide to SQL: Reference for Version 11.50xC6.

Overview	Reference
Load and Unload Data with External Tables Informix supports external tables. You can read and write from a source that is external to the database server. External tables provide an SQL interface to data in text files managed by the operating system or to data from a FIFO device. To create external tables, use the CREATE EXTERNAL TABLE statement. Use the existing DROP TABLE statement to drop an external table.	"SYSEXTDFILES" on page 1-25 "SYSEXTCOLS" on page 1-24 "SYSEXTERNAL" on page 1-26 Also see *IBM Informix Guide to SQL: Syntax* and *IBM Informix Administrator's Guide*.

Table 3. What's New in IBM Informix Guide to SQL: Reference for Version 11.50.xC5.

Overview	Reference
Logical Character Semantics in Character Type declarations In database locales that support multibyte code sets, such as **UTF-8**, a single logical character can occupy up to four bytes of storage. To prevent multibyte character strings from being truncated, you can use the new SQL_LOGICAL_CHAR configuration parameter to instruct the SQL parser to interpret the declared size of character data types in units of logical characters. By default, any explicit or default size specifications are interpreted in units of bytes.	"Character Data Types" on page 2-35

Table 4. What's New in IBM Informix Guide to SQL: Reference for Version 11.50.xC4

Overview	Reference
IFX_LARGE_PAGES Environment Variable (AIX®, Solaris) The **IFX_LARGE_PAGES** environment variable can enable the use of large pages for non-message shared memory segments that are resident in physical memory. The DBSA must set the RESIDENT configuration parameter accordingly, and must use operating system commands to configure a pool of these large pages. Informix can then use large pages from that pool, if they are available, for shared virtual memory segments.	"IFX_LARGE_PAGES" on page 3-40

Table 5. What's New in IBM Informix Guide to SQL: Reference for Version 11.50.xC1

Overview	Reference
BIGINT and BIGSERIAL data types This release introduces two data types, BIGINT and BIGSERIAL. These data types have the same ranges as the existing INT8 and SERIAL8 data types, and have storage and computational advantages.	"BIGINT" on page 2-5 "BIGSERIAL" on page 2-5

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 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.

All of the product documentation (including release notes, machine notes, and documentation notes) is available from the information center on the web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

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.

The IBM Informix Geodetic DataBlade® Module supports a subset of the data types from the *Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173*, as referenced by the document *Content Standard for Geospatial Metadata*, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

IBM Informix Dynamic Server (IDS) Enterprise Edition, Version 11.50 is certified under the Common Criteria. For more information, see *Common Criteria Certification: Requirements for IBM Informix Dynamic Server*, which is available at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SC23-7690-00.

Syntax diagrams

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

Table 6. Syntax Diagram Components

Component represented in PDF	Component represented in HTML	Meaning
	`>>----------`	Statement begins.
	`---------->`	Statement continues on next line.
	`>----------`	Statement continues from previous line.
	`----------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+------------------+---` `  '------LOCAL------'`	Optional item.

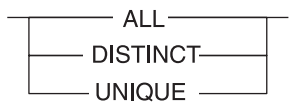
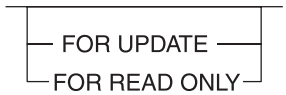
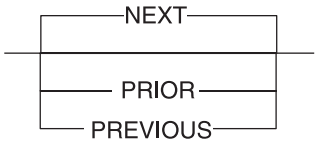
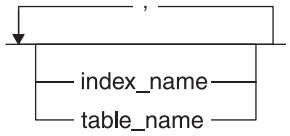
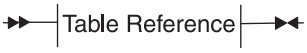
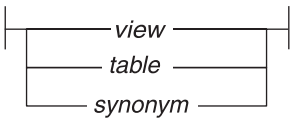
	`---+-----ALL-------+---`
	`  +--DISTINCT-----+`
	`  '---UNIQUE------'`
		Required item with choice. Only one item must be present.

Table 6. Syntax Diagram Components (continued)

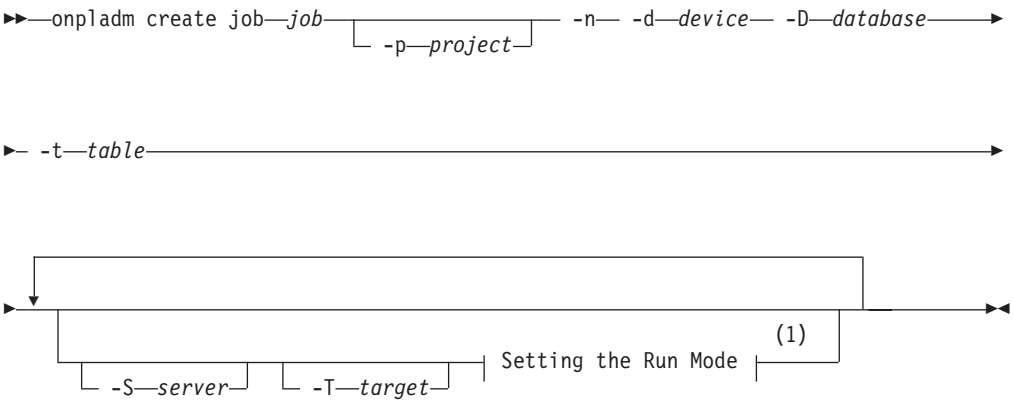
Component represented in PDF	Component represented in HTML	Meaning
	```---+------------------+--- +--FOR UPDATE-----+ '--FOR READ ONLY--'```	Optional items with choice are shown below the main line, one of which you might specify.
	```.---NEXT---------. ----+----------------+--- +---PRIOR--------+ '---PREVIOUS-----'```	The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.
	```.-------,-----------. V | ---+------------------+--- +---index_name---+ '---table_name---'```	Optional items. Several items are allowed; a comma must precede each repetition.
	```>>-| Table Reference |-><```	Reference to a syntax segment.
Table Reference 	```Table Reference |--+-----view--------+--| +------table------+ '----synonym------'```	Syntax segment.

How to read a command-line syntax diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a no-conversion job



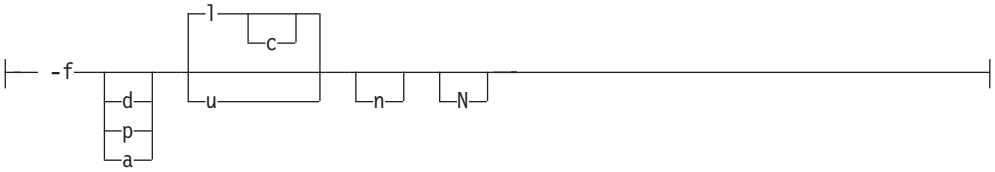
Notes:

1 See page Z-1

This diagram has a segment named "Setting the Run Mode," which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you

would find this segment on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the run mode:



To see how to construct a command correctly, start at the upper left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case-sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case-sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

Keywords are words reserved for statements and all commands except system-level commands.

When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

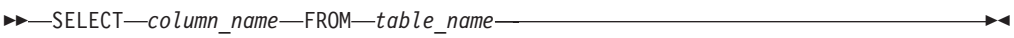
You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to provide documentation feedback

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

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- Go to the information center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp and open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.
- Add comments to topics directly in the Informix information 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! Find out more at http://publib.boulder.ibm.com/infocenter/idshelp/v115/topic/com.ibm.start.doc/contributing.htm.

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support website 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.

In This Chapter

This chapter provides 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, and synonyms
- Columns, constraints, indexes, and fragments
- Triggers
- Procedures, functions, routines, and associated messages
- Authorized users, roles, and privileges to access database objects
- Data types and casts
- Aggregate functions
- Access methods and operator classes
- Sequence objects
- External optimizer directives
- Inheritance relationships

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 1-47 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

Column Name	First Row	Second Row
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		
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		
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. 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** table holds 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 structure of an IBM Informix database. Here X indicates whether Informix, XPS, or both support the table.

System Catalog Table	XPS	Informix
"SYSAGGREGATES" on page 1-9		X
"SYSAMS" on page 1-9		X
"SYSATTRTYPES" on page 1-11		X
"SYSBLOBS" on page 1-12	X	X
"SYSCASTS" on page 1-12		X
"SYSCHECKS" on page 1-13	X	X
"SYSCHECKUDRDEP" on page 1-14		X
"SYSCOLATTRIBS" on page 1-14		X
"SYSCOLAUTH" on page 1-15	X	X
"SYSCOLDEPEND" on page 1-15	X	X
"SYSCOLUMNS" on page 1-16	X	X
"SYSCONSTRAINTS" on page 1-20	X	X
"SYSDEFAULTS" on page 1-20	X	X
"SYSDEPEND" on page 1-21	X	X
"SYSDIRECTIVES" on page 1-22		X
"SYSDISTRIB" on page 1-22	X	X
"SYSDOMAINS" on page 1-23		X
"SYSERRORS" on page 1-23		X
"SYSEXTCOLS (XPS)" on page 1-25	X	
"SYSEXTDFILES (XPS)" on page 1-26	X	
"SYSEXTERNAL (XPS)" on page 1-26	X	
"SYSFRAGAUTH" on page 1-27		X
"SYSFRAGMENTS" on page 1-28	X	X
"SYSINDEXES" on page 1-29	X	X
"SYSINDICES" on page 1-30		X
"SYSINHERITS" on page 1-31		X
"SYSLANGAUTH" on page 1-32		X
"SYSLOGMAP" on page 1-32		X
"SYSNEWDEPEND (XPS)" on page 1-32	X	
"SYSOBJSTATE" on page 1-33		X
"SYSOPCLASSES" on page 1-34		X
"SYSOPCLSTR" on page 1-34	X	X
"SYSPROCAUTH" on page 1-35	X	X
"SYSPROCCOLUMNS" on page 1-37	X	X

System Catalog Table	XPS	Informix
"SYSPROCBODY" on page 1-36	X	X
"SYSPROCEDURES" on page 1-37	X	X
"SYSPROCPLAN" on page 1-40	X	X
"SYSREFERENCES" on page 1-40	X	X
"SYSREPOSITORY (XPS)" on page 1-41	X	
"SYSROLEAUTH" on page 1-41	X	X
"SYSROUTINELANGS" on page 1-42		X
"SYSSECLABELCOMPONENTS" on page 1-42		X
"SYSSECLABELCOMPONENTELEMENTS" on page 1-42		X
"SYSSECPOLICIES" on page 1-43		X
"SYSSECPOLICYCOMPONENTS" on page 1-43		X
"SYSSECPOLICYEXEMPTIONS" on page 1-44		X
"SYSSECLABELS" on page 1-45		X
"SYSSECLABELNAMES" on page 1-44		X
"SYSSECLABELAUTH" on page 1-44		X
"SYSSEQUENCES" on page 1-45		X
"SYSSYNONYMS" on page 1-45	X	X
"SYSSYNTABLE" on page 1-46	X	X
"SYSTABAMDATA" on page 1-46		X
"SYSTABAUTH" on page 1-47	X	X
"SYSTABLES" on page 1-47	X	X
"SYSTRACECLASSES" on page 1-50		X
"SYSTRACEMSGS" on page 1-50		X
"SYSTRIGBODY" on page 1-51	X	X
"SYSTRIGGERS" on page 1-52	X	X
"SYSUSERS" on page 1-52	X	X
"SYSVIEWS" on page 1-53	X	X
"SYSVIOLATIONS" on page 1-53	X	X
"SYSXADATASOURCES" on page 1-54		X
"SYSXASOURCETYPES" on page 1-54		X
"SYSXTDDESC" on page 1-55		X
"SYSXTDTYPEAUTH" on page 1-55		X
"SYSXTDTYPES" on page 1-55		X

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

SYSAGGREGATES

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

Column	Type	Explanation
name	VARCHAR(128)	Name of the aggregate
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 that is described in the *IBM Informix Guide to SQL: Syntax*. The **sysams** table has the following columns.

Column	Type	Explanation
am_name	VARCHAR(128)	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 *or* a = all types: extspaces, dbspaces, and sbspaces. If the access method is not user-defined (that is, if it is built in or registered during database creation by the server), it supports dbspaces. D *or* d = dbspaces only S *or* s = sbspaces only (smart-large-object space) X *or* x = extspaces only
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.

Column	Type	Explanation
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 in order 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	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

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

For each of the nearly 20 columns that follow **am_costfactor**, the value is the **sysprocedures.procid** value for the corresponding routine.

The **am_sptype** column can have multiple entries. For example:

- `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.
- `DS` means the access method supports dbspaces and sbspaces.
- `sx` means the access method supports sbspaces and extspaces.

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.

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 1-55).
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)

Column	Type	Explanation
type	SMALLINT	Code for the data type See the description of **syscolumns.coltype** (page "SYSCOLUMNS" on page 1-16).
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 1-55).

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.

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.

Column	Type	Explanation
spacename	VARCHAR(128)	Name of partition, dbspace, or family
type	CHAR(1)	Code identifying the type of storage media: M = Magnetic O = Optical
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.

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

Column	Type	Explanation
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
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.

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.

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 listed in the PUT clause of the CREATE TABLE statement.

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)	A record is kept of the most recent access of this smart-large-object column by a user.
		LO_NOKEEP_LASTACCESS_TIME (0x00001000 = 8)	No record is kept of the most recent access of this smart-large-object column by a user.
		HI_INTEG (0x00010000= 16)	Data pages have headers and footers to detect incomplete writes and data corruption.
		MODERATE_INTEG (Not available at this time)	Data pages do not have headers and 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
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 and 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 1-14, which lists every check constraint that is referenced by a user-defined routine.

See also the **sysnewdepend** table in "SYSNEWDEPEND (XPS)" on page 1-32, which describes the column dependencies of generalized-key indexes.

See also the **sysreferences** table in "SYSREFERENCES" on page 1-40, 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.

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).
coltype	SMALLINT	Code indicating the data type of the column: 0 = CHAR 1 = SMALLINT 2 = INTEGER 3 = FLOAT 4 = SMALLFLOAT 5 = DECIMAL 6 = SERIAL * 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 = Variable-length opaque type [2] 41 = Fixed-length opaque type [2] 43 = LVARCHAR (client-side only) 45 = BOOLEAN 52 = BIGINT 53 = BIGSERIAL 2061 = IDSSECURITYLABEL [2] 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

Column	Type	Explanation
seclabid	INTEGER	The label ID of the security label associated with the column if it is a protected column. NULL otherwise.

Notes:

[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] See Opaque Data Types for more information.

Extended Parallel Server does not support opaque data types, nor the complex data types SET, MULTISET, LIST, unnamed and named ROW.

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.

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 1-16.

Opaque Data Types

There are specific data types that are implemented by the database server as built-in opaque data types. The type definition for a built-in opaque data type is provided by the database server.

The built-in opaque types do not have a unique **coltype** value. Instead, the **coltype** values are based on the category of opaque type. The following table lists the **coltype** values for the built-in opaque data types:

Category of Opaque Data Type	Predefined Data Type	Value for coltype Column
Fixed-length opaque type	BLOB, BOOLEAN, and CLOB	41
Variable-length opaque type	LVARCHAR	40
DISTINCT of VARCHAR(128)	IDSSECURITYLABEL	2061

The different fixed-length opaque types are distinguished by the **extended_id** column in the **sysxtdtypes** system catalog table.

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 , 8 (XPS)
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.

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:

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.

Column	Type	Explanation
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. For Extended Parallel Server, this index omits the **class** column.

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.

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:

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
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.

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	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
type	CHAR(1)	Type of statistics: A = **encdat** has ASCII-encoded histogram in fixed-length character field S = **encdat** has user-defined statistics

Column	Type	Explanation
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.

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 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.

SYSDOMAINS

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

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 more information about these messages, see http://publib.boulder.ibm.com/infocenter/idshelp/v115/topic/com.ibm.em.doc/errors_ids115.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 concepts

Using the SQLSTATE Error Status Code (SQL Syntax)

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.

SYSEXTCOLS (XPS)

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
picture	CHAR(256)	Reserved for future use
decimal	SMALLINT	Precision for external decimals
extstype	VARCHAR(128)	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

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(598)	Absolute source or target file path
blobdir	CHAR(473)	Absolute or relative directory name
clobdir	CHAR(473)	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.

SYSEXTDFILES (XPS)

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(152)	Data file entry

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)
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.

SYSEXTERNAL (XPS)

For each external table, a single row exists in the **sysexternal** system catalog table. The **tabid** column associates the external table in this system catalog table with an entry in **systables**.

Column	Type	Explanation
tabid	INTEGER	Unique identifying code of an external table

Column	Type	Explanation
fmttype	CHAR(1)	Type of format: D = (delimited) F = (fixed) I = (IBM Informix)
codeset	VARCHAR(128)	ASCII, EBCDIC
recdelim	CHAR(4)	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 per coserver
rejectfile	CHAR(128)	Name of reject file
flags	INTEGER	Optional **load** flags
ndfiles	INTEGER	Number of data files in **sysextdfiles**

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 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.

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
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---

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

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
partn	INTEGER	Identifying code of physical storage location
strategy	CHAR(1)	Code for type of fragment distribution strategy: • R = Round-robin fragmentation strategy • E = Expression-based fragmentation strategy • I = IN DBSPACE clause specifies a storage location as part of fragmentation strategy • T = Table-based fragmentation 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
evalpos	INTEGER	Position of fragment in the fragmentation list.
exprtext	TEXT	Expression for fragmentation strategy
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, **npused** is the number of data pages. For index-fragmentation strategies, **npused** is the number of leaf pages.
nrows	FLOAT	For tables, **nrows** is the number of rows in the fragment. For indexes, **nrows** is the number of unique keys.
clust	FLOAT	Degree of index clustering; smaller numbers correspond to greater clustering.

Column	Type	Explanation
partition	Name of the fragment	Can match the name of the IBM Informix dbspace that stores the fragment, or it can be a different name

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

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.)

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.

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 (XPS) 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
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

Column	Type	Explanation
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 Normal = 0x00000001 (XPS) Coarse = 0x00000002 (XPS)

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.

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

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 1-1. **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)	Index type U = Unique D = Duplicates allowed
clustered	CHAR(1)	Clustered or nonclustered index (C = Clustered)
levels	SMALLINT	Number of tree levels
leaves	FLOAT	Number of leaves
nunique	FLOAT	Number of unique keys in the first column

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

Column	Type	Explanation
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	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.)
amparam	LVARCHAR	List of parameters used to customize the behavior of the **amid** access method
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

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 1-29.

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.

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

Column	Type	Explanation
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).

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
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.

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.

SYSNEWDEPEND (XPS)

The **sysnewdepend** system catalog table contains information about generalized-key indexes that are unavailable in the **sysindexes** table. The dependencies between a generalized-key index and the tables in the FROM clause of the CREATE INDEX statement are stored in the **sysnewdepend** table, which has the following columns.

Column	Type	Explanation
scrid1	VARCHAR(128)	Name of the generalized-key index

Column	Type	Explanation
scrid2	INTEGER	Unique identifying code (= **tabid**) of the indexed table
type	INTEGER	Code for the type of generalized-key index
destid1	INTEGER	The **systables.tabid** value for the table on which the generalized-key index depends
destid2	INTEGER	The column number within the **destid1** table

A composite index on the **scrid1**, **scrid2**, and **type** columns allows duplicate values. Another composite index on the **destid1**, **destid2**, and **type** columns also allows 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.

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	List of names of the operators that belong to this operator class
support	LVARCHAR	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.

SYSOPCLSTR

The **sysopclstr** system catalog table defines each optical cluster in the database. It contains one row for each optical cluster. The **sysopclstr** table has the following columns.

Column	Type	Explanation
owner	VARCHAR(32)	Name of the owner of the optical cluster
clstrname	VARCHAR(128)	Name of the optical cluster
clstrsize	INTEGER	Size of the optical cluster
tabid	INTEGER	Unique identifying code for the table
blobcol1	SMALLINT	BYTE or TEXT column number 1
blobcol2	SMALLINT	BYTE or TEXT column number 2
blobcol3	SMALLINT	BYTE or TEXT column number 3
blobcol4	SMALLINT	BYTE or TEXT column number 4
blobcol5	SMALLINT	BYTE or TEXT column number 5
blobcol6	SMALLINT	BYTE or TEXT column number 6
blobcol7	SMALLINT	BYTE or TEXT column number 7
blobcol8	SMALLINT	BYTE or TEXT column number 8
blobcol9	SMALLINT	BYTE or TEXT column number 9
blobcol10	SMALLINT	BYTE or TEXT column number 10

Column	Type	Explanation
blobcol11	SMALLINT	BYTE or TEXT column number 11
blobcol12	SMALLINT	BYTE or TEXT column number 12
blobcol13	SMALLINT	BYTE or TEXT column number 13
blobcol14	SMALLINT	BYTE or TEXT column number 14
blobcol15	SMALLINT	BYTE or TEXT column number 15
blobcol16	SMALLINT	BYTE or TEXT column number 16
clstrkey1	SMALLINT	Cluster key number 1
clstrkey2	SMALLINT	Cluster key number 2
clstrkey3	SMALLINT	Cluster key number 3
clstrkey4	SMALLINT	Cluster key number 4
clstrkey5	SMALLINT	Cluster key number 5
clstrkey6	SMALLINT	Cluster key number 6
clstrkey7	SMALLINT	Cluster key number 7
clstrkey8	SMALLINT	Cluster key number 8
clstrkey9	SMALLINT	Cluster key number 9
clstrkey10	SMALLINT	Cluster key number 10
clstrkey11	SMALLINT	Cluster key number 11
clstrkey12	SMALLINT	Cluster key number 12
clstrkey13	SMALLINT	Cluster key number 13
clstrkey14	SMALLINT	Cluster key number 14
clstrkey15	SMALLINT	Cluster key number 15
clstrkey16	SMALLINT	Cluster key number 16

The contents of this table are sensitive to CREATE OPTICAL CLUSTER, ALTER OPTICAL CLUSTER, and DROP OPTICAL CLUSTER statements that have been executed on databases that support optical cluster subsystems. Changes that affect existing optical clusters are reflected in this table only after you run the UPDATE STATISTICS statement.

A composite index on the **clstrname** and **owner** columns allows only unique values. A simple index on the **tabid** column allows duplicate 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.

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

Column	Type	Explanation
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.

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
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.

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
pxid	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.

Extended Parallel Server

For Extended Parallel Server, **sysprocedures** has the following columns.

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 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 instruction code

Column	Type	Explanation
numargs	INTEGER	Number of arguments to routine

A composite index on **procname** and **owner** requires unique values.

IBM Informix

For IBM Informix **sysprocedures** has the following columns.

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)

Column	Type	Explanation
commutator	VARCHAR(128)	Name of commutator function
negator	VARCHAR(128)	Name of the negator function
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

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.

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 (IDS) Text of the SPL routine (XPS)
collation	CHAR(32)	Collating order at the time when routine was created

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.

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
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.

SYSREPOSITORY (XPS)

The **sysrepository** system catalog table contains data about generalized-key indexes that the **sysindexes** system catalog table does not provide.

Column	Type	Explanation
id1	VARCHAR(128)	Index from the generalized-key (GK) index
id2	INTEGER	Tabid of table with the generalized-key index
type	INTEGER	Integer code for type of object In this release, the only value that can be shown is 1, indicating a GK index type.
seqid	SERIAL	Reserved for future use (This value is not related to **syssequences.seqid**.)
desc	TEXT	The CREATE INDEX statement of a GK index
bin	BYTE	Internal representation of the generalized-key index

The contents of the **sysrepository** table are useful when a generalized-key index must be rebuilt during a recovery or if a user wants to see the CREATE statement for a specific generalized-key index.

The **desc** column contains the CREATE statement for each generalized-key index in the database.

An index on the **seqid** column allows duplicate values. A composite index on the **id1**, **id2**, and **type** columns requires unique combinations of 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.

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.

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.

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
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
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

SYSSECLABELAUTH

The **sysseclabelauth** system catalog table records the 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

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

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
cache	INTEGER	Number of preallocated values in sequence cache
order	CHAR(1)	Zero means NOORDER, 1 means ORDER

SYSSYNONYMS

The **syssynonyms** system catalog table lists the synonyms for each table or view. Except for database servers that have migrated from certain interim releases of Version 1.10 IBM Informix, only the **syssyntable** table describes synonyms, and the **syssynonyms** table is unused. It has the following columns.

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.

Column	Type	Explanation
tabid	INTEGER	Identifying code of the table
am_param	CHAR(256)	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.

Column	Type	Server	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
tabauth	CHAR(9) CHAR(8)	Informix, XPS	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 1-29). (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.

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

Column	Type		Explanation
nindexes	SMALLINT		Number of indexes on the table
nrows	FLOAT		Number of rows in the table
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 level • P = Page level • R = Row level • T = Table level (XPS)
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: • ST_RAW = 0x00000010 (Informix) • RAW = 0x00000002 (XPS) • STATIC = 0x00000004 (XPS) • OPERATIONAL = 0x00000008 (XPS) • STANDARD = 0x00000010 (XPS) • EXTERNAL = 0x00000020 (XPS)
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
am_id	INTEGER		Access method code (key to **sysams** table) NULL or 0 indicates built-in storage manager
minrowsize	SMALLINT	XPS	Minimum row size in bytes

Column	Type		Explanation
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

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.

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.

Column	Type	Explanation
name	VARCHAR(128)	Name of the message
msgid	SERIAL	Identifying code of the message template

Column	Type	Explanation
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.

Column	Type	Explanation
trigid	INTEGER	Identifying code of the trigger
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
collation	CHAR(32)	Collating order at the time when trigger was created

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.

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

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:

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.

Column	Type	Explanation
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
viewtext	CHAR(64)	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 (Informix). The maximum number of rows allowed in the violations table for each coserver (XPS)

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.

Extended Parallel Server does not use the diagnostic table when a constraint violation occurs. Rather, the database server stores additional information in the violations table. The violations table contains the data that the transaction refused and an indication of the cause.

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_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 **sysxtdtype** 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.

Column	Type	Explanation
extended_id	SERIAL	Unique identifying code for extended data type
domain	CHAR(1)	Code for the domain of the UDT

Column	Type	Explanation
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 ' ' (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
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 1-16, 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.

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 an 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

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 an 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

In This Chapter

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.

This chapter describes 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 IBM Informix.

The following diagram shows the logical categories of data types that the IBM Informix supports. The shaded categories indicate the additional data types that are supported only on IBM Informix.

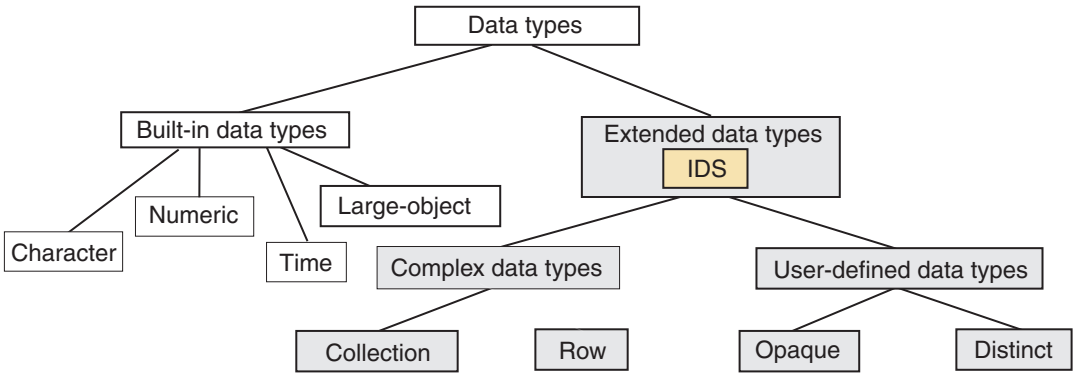


Figure 2-1. Overview of Supported Data Types

This diagram is simplified; some built-in types are implemented as opaque types, and are only supported on IBM Informix. That is, *opaque* and *built-in* are not disjunct categories, though most built-in data types are not opaque.

Built-in data types (which are system-defined) and *extended* data types (which you can define) share the following characteristics. You can:

- 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.

For exceptions, see the description of each data type. For an overview, see "Built-In Data Types" on page 2-35 and "Extended Data Types" on page 2-43.

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*.

Data Types That IBM Informix Supports

The following table lists all of the built-in data types that IBM Informix supports.

Table 2-1. Data Types That IBM Informix Supports

Data Type	Explanation
"BIGINT" on page 2-5	Stores 8-byte integer values from $-(2^{63} -1)$ to $2^{63} -1$
"BIGSERIAL" on page 2-5	Stores sequential, 8-byte integers from 1 to $2^{63} -1$
"BYTE" on page 2-7	Stores any kind of binary data, up to 2^{31} bytes in length
"CHAR(n)" on page 2-8	Stores character strings; collation is in code-set order
"CHARACTER(n)" on page 2-10	Is a synonym for CHAR
"CHARACTER VARYING(m,r)" on page 2-10	Stores character strings of varying length (ANSI-compliant); collation is in code-set order
"DATE" on page 2-11	Stores calendar dates
"DATETIME" on page 2-11	Stores calendar date combined with time of day
"DEC" on page 2-14	Is a synonym for DECIMAL
"DECIMAL" on page 2-14	Stores floating-point numbers with definable precision; if database is ANSI-compliant, the scale is zero
"DECIMAL (p,s) Fixed Point" on page 2-15	Stores fixed-point numbers of defined scale and precision
"DOUBLE PRECISION" on page 2-17	Synonym for FLOAT
"FLOAT(n)" on page 2-17	Stores double-precision floating-point numbers corresponding to the **double** data type in C
"INT" on page 2-18	Is a synonym for INTEGER
"INT8" on page 2-18	Stores 8-byte integer values from $-(2^{63} -1)$ to $2^{63} -1$
"INTEGER" on page 2-18	Stores whole numbers from -2,147,483,647 to +2,147,483,647
"INTERVAL" on page 2-18	Stores a span of time (or level of effort) in units of *years* and *months*.
"INTERVAL" on page 2-18	Stores a span of time in a contiguous set of units of *days*, *hours*, *minutes*, *seconds*, and *fractions of a second*
"MONEY(p,s)" on page 2-22	Stores currency amounts
"NCHAR(n)" on page 2-24	Same as CHAR, but can support localized collation
"NUMERIC(p,s)" on page 2-24	Synonym for DECIMAL(*p,s*)
"NVARCHAR(m,r)" on page 2-24	Same as VARCHAR, but can support localized collation

Table 2-1. Data Types That IBM Informix Supports (continued)

Data Type	Explanation
"REAL" on page 2-25	Is a synonym for SMALLFLOAT
"SERIAL(n)" on page 2-28	Stores sequential integers (> 0) in positive range of INT
"SERIAL8(n)" on page 2-29	Stores sequential integers (> 0) in positive range of INT8
"SMALLFLOAT" on page 2-31	Stores single-precision floating-point numbers corresponding to the **float** data type of the C language
"SMALLINT" on page 2-31	Stores whole numbers from -32,767 to +32,767
"TEXT data type" on page 2-32	Stores any kind of text data, up to 2^{31} bytes in length
"VARCHAR(m,r)" on page 2-33	Stores character strings of varying length (up to 255 bytes); collation is in code-set order

These built-in SQL data types are valid in all IBM 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

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

Additional Data Types that IBM Informix supports

The following table lists the additional data types that IBM Informix supports.

Table 2-2. Additional Data Types That IBM Informix Supports

Data Type	Explanation
"BLOB" on page 2-6	Stores binary data in random-access chunks
binary18	Stores 18 byte binary-encoded strings
binaryvar	Stores binary-encoded strings with a maximum length of 255 bytes
"BOOLEAN" on page 2-7	Stores Boolean values true and false
"CLOB" on page 2-10	Stores text data in random-access chunks
"DISTINCT" on page 2-16	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	Stores a calendar for a TimeSeries data type
CalendarPattern	Stores the structure of the calendar pattern for a Calendar data type
"IDSSECURITYLABEL" on page 2-17	Stores LBAC security label objects.

Table 2-2. Additional Data Types That IBM Informix Supports (continued)

Data Type	Explanation
"LIST(e)" on page 2-20	Stores a sequentially ordered collection of elements, all of the same data type, *e*; allows duplicate values
lld_locator	Stores a large object identifier
lld_lob_data	Stores the location of a smart large object and specifies whether the object contains binary or character data
"LVARCHAR(m)" on page 2-22	Stores variable-length strings of up to 32,739 bytes
"MULTISET(e)" on page 2-23	Stores a non-ordered collection of values, with elements all of the same data type, *e*; allows duplicate values.
node	Stores a combination of integers and decimal points that represents hierarchical relationships, of variable length up to 256 characters
"OPAQUE" on page 2-25	Stores a user-defined data type whose internal structure is inaccessible to the database server
"ROW, Named" on page 2-25	Stores a named ROW type
"ROW, Unnamed" on page 2-26	Stores an unnamed ROW type
"SET(e)" on page 2-30	Stores a non-ordered collection of elements, all of the same data type, *e*; does not allow duplicate values
ST_LineString	Stores a one-dimensional object as a sequence of points defining a linear interpolated path
ST_MultiLineString	Stores a collection of ST_LineString data types
ST_MultiPoint	Stores a collection of ST_Point data types
ST_MultiPolygon	Stores a collection of ST_Polygon data types
ST_Point	Stores a zero-dimensional geometry that occupies a single location in coordinate space
ST_Polygon	Stores a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings
TimeSeries	Stores a collection of row subtypes

These extended data types of IBM 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 IBM 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 IBM 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 IBM Informix instances cannot return values of complex or smart large object data types, nor of most distinct or built-in opaque data types. Only the following data types 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.

Description of Data Types

This section describes the data types that IBM Informix supports.

BIGINT

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

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.

As is the case with all serial data types, the BIGSERIAL data type stores the negative values that you provide. However, the generated value is always a positive number, from 1 to 2^{63} -1.

The BIGSERIAL data type can store values 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.

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)" on page 2-28
- 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

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 term *smart large object* refers to BLOB and CLOB data types. Use CLOB data types (see page "CLOB" on page 2-10) for random access to text data. For general information about BLOB and CLOB data types, see "Smart Large Objects" on page 2-38.

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 reference

FILETOBLOB and FILETOCLOB Functions (SQL Syntax)

LOTOFILE Function (SQL Syntax)

LOCOPY Function (SQL Syntax)

BOOLEAN

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

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 BYTE and TEXT data types. 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 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 items 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

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 insert data into BYTE columns in the following ways:

- 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)

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. (For more information about East Asian locales that support multibyte code sets, see "Multibyte Characters with VARCHAR " on page 2-34.)

A CHAR(*n*) column has a length of *n* bytes, where $1 \leq n \leq 32{,}767$. 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 2-36.

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 2-35.) 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)

The CHARACTER data type is a synonym for CHAR.

CHARACTER VARYING(m,r)

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)" on page 2-33.

CLOB

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.

Use the BLOB data type (see "BLOB" on page 2-6) for random access to binary data. For general information about the CLOB and BLOB data types, see "Smart Large Objects" on page 2-38.

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

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" on page 3-19.

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

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 (see "INTERVAL" on page 2-18), 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 non-default precision for the *largest_qualifier* time unit.
- A DATETIME value that includes YEAR, MONTH, or both YEAR and MONTH. Time units can also include smaller time units, whereas an INTERVAL data type that stores days (or smaller time units) cannot store months or years.

The *largest_qualifier* and *smallest_qualifier* of a DATETIME data type can be any of the fields that Table 2-3 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 2-3. 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 to 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 to 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 Table 2-3 lists.

Write values for the field qualifiers as integers and separate them with delimiters. Table 2-4 on page 2-13 lists the delimiters that are used with DATETIME values in the default U.S. English locale. (These are a superset of the delimiters that are used in INTERVAL values; see Table 2-6 on page 2-20.)

Table 2-4. 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

Figure 2-2 shows a DATETIME YEAR TO FRACTION(3) value with delimiters.

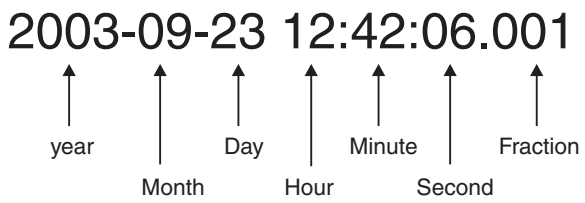


Figure 2-2. Example DATETIME Value with Delimiters

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. For information about setting **DBCENTURY**, see "DBCENTURY" on page 3-16.

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.

For information about how to use DATETIME values in arithmetic and relational expressions, see "Manipulating DATE with DATETIME and INTERVAL Values" on page 2-41. For more information about the DATETIME data type see the *IBM Informix Guide to SQL: Syntax*.

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.

With an ESQL API, the **DBTIME** environment variable also affects DATETIME formatting. Non-default 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.

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*.

For more information about **DBTIME**, see "DBTIME" on page 3-29. For more information about **DBCENTURY**, see "DBCENTURY" on page 3-16. For more information about locales and GLS environment variables that can specify end-user DATETIME formats, see the *IBM Informix GLS User's Guide*.

DEC

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, however, 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 first digit 1 byte for the decimal point 4 bytes for the rest of the digits in the declared 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 2-3 on page 2-16 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.

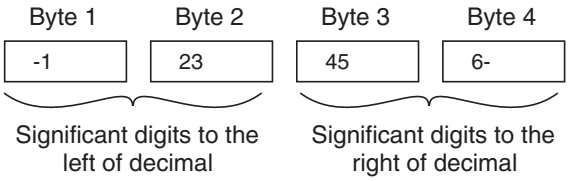


Figure 2-3. Schematic That Illustrates the Storage of Digits in a Decimal (p,s) Value

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

A DISTINCT type is a data type that is derived from one of the following source types (called the *base type*):

- 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 2-51. 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 2-36.

DOUBLE PRECISION

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

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.

IDSSECURITYLABEL

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. A table that has a security policy can have only one IDSSECURITYLABEL column. A table with no security policy can have none. You cannot encrypt the security label in a column of type IDSSECURITYLABEL.

INT

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. Extended Parallel Server stores INT8 values as 8 bytes.

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

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

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 Table 2-5 on page 2-19.

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 2-5. Interval Classes

Interval Class	Time Units	Valid Entry
YEAR-MONTH INTERVAL	YEAR	A number of years
	MONTH	A number of months
DAY-TIME INTERVAL	DAY	A number of days
	HOUR	A number of hours
	MINUTE	A number of minutes
	SECOND	A number of seconds
	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 $1 \leq n \leq 5$.

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 Table 2-5 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). Table 2-6 shows the INTERVAL delimiters.

Table 2-6. 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 $1 \leq n \leq 5$, 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 data in arithmetic and relational operations, see "Manipulating DATE with DATETIME and INTERVAL Values" on page 2-41. 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*.

LIST(e)

The LIST data type is a collection type that can store ordered non-NULL elements of the same SQL data type. It 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)" on page 2-23 and "SET(e)" on page 2-30. For complex data types that can store a set of values that includes different SQL data types, see "ROW Data Types" on page 2-45.

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)

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.

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 2-36.

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

MONEY(p,s)

The MONEY data type stores currency amounts. Like 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" on page 3-23.

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)

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)" on page 2-20.)

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, Named" on page 2-25.

NCHAR(n)

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 localized collation.

Size specifications ib NCHAR 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 2-36.

NUMERIC(p,s)

The NUMERIC data type is a synonym for fixed-point DECIMAL.

NVARCHAR(m,r)

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 localized collation. (The section "Collating VARCHAR Values" on page 2-35 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 feature that is described in the section "Logical Character Semantics in Character Type Declarations" on page 2-36.

No more than 195 columns of the same table can be NVARCHAR data types.

OPAQUE

An OPAQUE type is a data type for which you must provide the following information to the database server:

- 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

The REAL data type is a synonym for SMALLFLOAT.

ROW, 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 2-45.

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, 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)

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 can specify a positive or negative 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)

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 2-18).

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table.

- You can specify a positive or negative 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 2-6

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 2-6

SET(e)

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)" on page 2-23.)

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)" on page 2-20.) 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 4 bytes of storage. Conversion of a SMALLFLOAT value to a DECIMAL value results in 9 digits of precision.

SMALLINT

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 the TEXT and BYTE data types.

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 values 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 insert data into TEXT columns in the following ways:

- 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.

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 2-9.

Unnamed ROW

See "ROW, Unnamed" on page 2-26.

VARCHAR(m,r)

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" on page 3-45 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 2-9.

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.

Multibyte Characters with VARCHAR

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*.

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 2-36.

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
Numeric	BIGINT, BIGSERIAL, DECIMAL, FLOAT, INT8, INTEGER, MONEY, SERIAL, SERIAL8, SMALLFLOAT, SMALLINT
Time	DATE, DATETIME, INTERVAL

Extended Parallel Server does not support BLOB, CLOB, IDSSECURITYLABEL, or LVARCHAR. For a description of character, numeric, and miscellaneous data types, see the appropriate entry in "Description of Data Types" on page 2-5. Page references are in the alphabetic list in Table 2-1 on page 2-2.

Sections that follow provide additional information about character, large-object, and time data types.

Character Data Types

The character data types store string values.

Built-in Character Types

Table 2-7. Attributes of Built-In Character Data Types

	Server	Size (in bytes)	Default	Reserved	Collation	Length
CHAR(n)	Informix, XPS	1 to 32,767	1 byte	None	Code set	Fixed
NCHAR(n)	Informix, XPS	1 to 32,767	1 byte	None	Localized	Fixed
VARCHAR(m, r)	Informix, XPS	1 to 256	0 for **r**	0 to 255 bytes	Code set	Variable
NVARCHAR(m, r)	Informix, XPS	1 to 256	0 for **r**	0 to 255 bytes	Localized	Fixed

Table 2-7. Attributes of Built-In Character Data Types (continued)

	Server	Size (in bytes)	Default	Reserved	Collation	Length
LVARCHAR(m)	Informix	1 to 32,739	2048 bytes	None	Code set	Variable

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 256) in their declaration. For VARCHAR or NVARCHAR columns on which an index is defined, the maximum size is 255 bytes.

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.

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 2-4 shows the large-object data types.

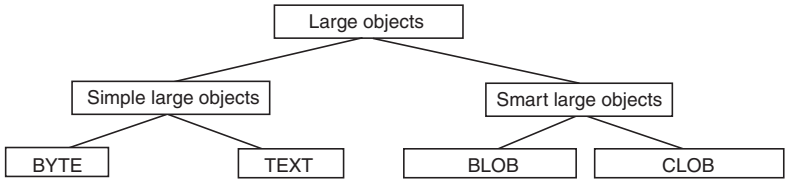


Figure 2-4. Large-Object Data Types

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" on page 2-7.

TEXT Stores text data. For more detailed information about this data type, see the description on page "TEXT data type" on page 2-32.

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 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 2-46.) 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" on page 2-6.

CLOB Stores text data. For more information about this data type, see "CLOB" on page 2-10.

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 log 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 page headers to detect data corruption

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 an IBM Informix ESQL/C program, 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 2-8 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 2-8. 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
DATETIME	-	CURRENT	INTERVAL
CURRENT	-	DATETIME	INTERVAL
INTERVAL	+	CURRENT	DATETIME
CURRENT	+ *or* -	INTERVAL	DATETIME
DATETIME	+ *or* -	UNITS	DATETIME

Table 2-8. Arithmetic Operations on DATE, DATETIME, and INTERVAL Values (continued)

Operand 1	Operator	Operand 2	Result
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]
```

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 2-9 shows.

Table 2-9. 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 2-9 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" on page 3-19 and "DBTIME" on page 3-29. 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.

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.

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*.

Complex Data Types

A *complex data type* can store one or more values of other built-in or extended data types.

Figure 2-5 shows the complex types that IBM Informix supports.

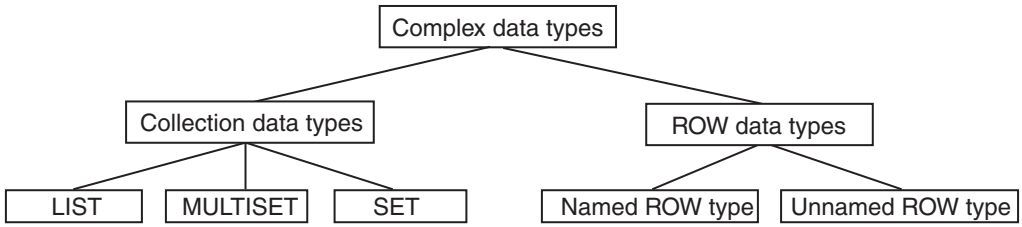


Figure 2-5. Complex Data Types of IBM Informix

The following table summarizes the structure of the complex data types.

Data Type		Description
Collection types:		Complex data types that are made up of elements, each of which is of the same data type.
	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.
	SET	A group of elements, each of which is unique. The order of the elements is ignored.
ROW types:		Complex data types that are made up of fields.
	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	**export**	**LO_handles**
output	**import_binary**	**hash**
send	**export_binary**	**lessthan**
recv	**assign**	**equal**
import	**destroy**	**lessthan** (for ROW types only)

Sections 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 an IBM Informix ESQL/C program

  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, Named" on page 2-25 and "ROW, Unnamed" on page 2-26.

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" on page 2-16. See also *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Opaque Data Types

An opaque data type is a user-defined 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 types, UDTs whose source types are built-in types are opaque data types.

The built-in data types BLOB, BOOLEAN, CLOB, and LVARCHAR are implemented as opaque data types. You cannot access these built-in opaque data types in cross-server distributed operations, but you can access them in other databases of the same IBM Informix instance.

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" on page 2-25. 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 1-12.

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.

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

Legend:

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

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, IMPEX, or IMPEXBIN). 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, IMPEX, or IMPEXBIN) and then explicitly cast this type to the second opaque type.

For information about casting and the IMPEX, IMPEXBIN, LVARCHAR, and SENDRECV 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.

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)

Target Type	Opaque Type	Distinct Type	Named ROW Type	Unnamed ROW Type	Collection Type	Built-in Type

[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

In This Chapter

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.

This chapter describes 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 and in the topic index in Table 3-4 on page 3-72, but are not explained in this publication.

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** (nor 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

The following sections discuss setting unsetting modifying and viewing environment variables. If you already use an IBM Informix product some or all of the appropriate environment variables might be set.

Where to Set Environment Variables on UNIX

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. See also "Using Environment Variables on UNIX."
- 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 IBM Informix, as described in the section "OPTCOMPIND environment variable" on page 3-61, and for environment variables of Extended Parallel Server that the **sysdbopen( )** or **sysdbclose( )** SPL routines can set. 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 (UNIX)" on page 3-34, 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 3-5, 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**

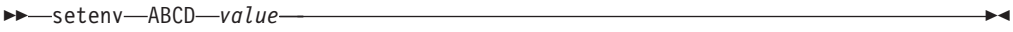
For Bourne shell or Korn shell
.profile

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.



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 an IBM Informix product, 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.

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 (UNIX)" on page 3-34.

Rules of Precedence

When an IBM Informix product 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 "Environment Settings."

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 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 IBM 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*.

Environment Settings

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

The following diagram shows the syntax for setting an environment variable at a command prompt in Windows.

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.

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 an IBM Informix product, 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

When an IBM Informix product accesses 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.

List of environment variables

The following table contains an alphabetic list of the environment variables that you can set for IBM Informix and SQL API products. Most of these environment variables are described in this section.

The notation *ERG* in the See column indicates an environment variable that must be set with the CDR_ENV configuration parameter and that is described in the appendix on configuration parameters and environment variables of the *IBM Informix Enterprise Replication Guide*.

The notation *GLS* in the See column indicates a GLS environment variable that is valid in nondefault locales and that is described in the GLS environment variables chapter of *IBM Informix GLS User's Guide*.

The notation *JDBC* in the See column indicates an environment variable that can be set in client environments that run applications with the IBM Informix JDBC Driver. These are described in the *IBM Informix JDBC Driver Guide*. For server-side JDBC, you should use property settings in the database URL rather than set server environment variables, because those environment settings would apply to all programs running on the database server.

Table 3-1. Alphabetical list of environment variables

Environment variable	XPS	Informix	Restrictions	See
"AC_CONFIG" on page 3-13	X	X	ON-Bar	
"ANSIOWNER" on page 3-14		X	None	
BIG_FET_BUF_SIZE				JDBC
CC8BITLEVEL			ESQL/C only	GLS
CDR_ATSRISNAME_DELIM		X	ER only	ERG
CDR_DISABLE_SPOOL		X	ER only	ERG
CDR_LOGDELTA		X	ER only	ERG
CDR_PERFLOG		X	ER only	ERG
CDR_ROUTER		X	ER only	ERG
CDR_RMSCALEFACT		X	ER only	ERG
CLIENT_LOCALE	X	X	None	GLS
"CPFIRST" on page 3-14	X	X	ESQL/C only	
CSM				JDBC
"DBACCNOIGN" on page 3-15	X	X	DB-Access only	
"DBANSIWARN" on page 3-16	X	X	None	JDBC
"DBBLOBBUF" on page 3-16	X	X	UNLOAD only	
"DBCENTURY" on page 3-16			SQL APIs only	
"DBDATE" on page 3-19	X	X	None	GLS
"DBDELIMITER" on page 3-21	X	X	None	
"DBEDIT" on page 3-21	X	X	None	
"DBFLTMASK" on page 3-22	X	X	DB-Access only	
"DBLANG" on page 3-22	X	X	None	GLS
"DBMONEY" on page 3-23	X	X	None	GLS

Table 3-1. Alphabetical list of environment variables (continued)

Environment variable	XPS	Informix	Restrictions	See
"DBONPLOAD" on page 3-24		X	HPL only	
"DBPATH" on page 3-24	X	X	None	
"DBPRINT" on page 3-26	X	X	UNIX only	
"DBREMOTECMD (UNIX)" on page 3-26	X	X	UNIX only	
"DBSPACETEMP" on page 3-27	X	X	None	JDBC
"DBTEMP" on page 3-28		X	DB-Access, Gateways	
"DBTIME" on page 3-29			SQL APIs only	GLS
"DBUPSPACE" on page 3-31	X	X	None	JDBC
DB_LOCALE	X	X	None	GLS
"DEFAULT_ATTACH environment variable" on page 3-32		X	Deprecated	
"DELIMIDENT environment variable" on page 3-33	X	X	None	JDBC
ENABLE_CACHE_TYPE				JDBC
ENABLE_HDRSWITCH				JDBC
"ENVIGNORE (UNIX)" on page 3-34	X	X	UNIX only	
ESQLMF	X	X	ESQL/C only	GLS
"FET_BUF_SIZE" on page 3-34	X	X	SQL APIs, DB-Access only	JDBC
"GLOBAL_DETACH_INFORM (XPS)" on page 3-35	X		None	
GLS8BITFSYS	X	X	None	GLS
GL_DATE	X	X	None	GLS
GL_DATETIME	X	X	None	GLS
GL_USEGLU		X	None	GLS
"IBM_XPS_PARAMS (XPS)" on page 3-36	X		None	
"IFMX_CART_ALRM (XPS)" on page 3-36	X		None	
"IFMX_HISTORY_SIZE (XPS)" on page 3-37	X		DB-Access	
"IFMX_OPT_FACT_TABS (XPS)" on page 3-37	X		None	
"IFMX_OPT_NON_DIM_TABS (XPS)" on page 3-38	X		None	
IFX_AUTOFREE				JDBC
IFX_BATCHUPDATE_PER_SPEC				JDBC
IFX_CODESETLOB				JDBC
"IFX_DEF_TABLE_LOCKMODE" on page 3-38		X	None	
"IFX_DIRECTIVES" on page 3-39	X	X	None	JDBC
"IFX_EXTDIRECTIVES" on page 3-39	X	X	Set on the client only	JDBC
IFX_FLAT_UCSQ				JDBC
IFX_GET_SMFLOAT_AS_FLOAT				JDBC
IFX_ISOLATION_LEVEL				JDBC
"IFX_LARGE_PAGES" on page 3-40	X		UNIX only	
"IFX_LOB_XFERSIZE" on page 3-41		X	Set on the client only	

Table 3-1. Alphabetical list of environment variables (continued)

Environment variable	XPS	Informix	Restrictions	See
IFX_LOCK_MODE_WAIT				JDBC
"IFX_LONGID" on page 3-42	X	X	None	
"IFX_NETBUF_PVTPOOL_SIZE (UNIX)" on page 3-42	X	X	UNIX	
"IFX_NETBUF_SIZE" on page 3-43	X	X	None	
"IFX_NO_TIMELIMIT_WARNING" on page 3-44	X	X	None	
"IFX_NODBPROC" on page 3-44	X	X	None	
"IFX_NOT_STRICT_THOUS_SEP" on page 3-44		X	None	
"IFX_ONTAPE_FILE_PREFIX" on page 3-44		X	None	
"IFX_PAD_VARCHAR" on page 3-45		X	None	JDBC
IFX_SET_FLOAT_AS_SMFLOAT				JDBC
IFX_TRIMTRAILINGSPACES				JDBC
+ "IFX_UNLOAD_EILSEQ_MODE environment + variable" on page 3-45		X		
"IFX_UPDDESC" on page 3-46		X	None	
IFX_USEPUT				JDBC
IFX_XASPEC				JDBC
"IFX_XASTDCOMPLIANCE_XAEND" on page 3-46		X	None	JDBC
"IFX_XFER_SHMBASE" on page 3-47		X	None	
"IFX_XFER_SHMBASE" on page 3-47				
IFXHOST				JDBC
IFXHOST_SECONDARY				JDBC
"IFXRESFILE (Linux)" on page 3-47		X	Linux only	
"IMCCONFIG" on page 3-48		X		
"IMCSERVER" on page 3-48		X		
"INFORMIXC (UNIX)" on page 3-49			ESQL/C, UNIX only	
"INFORMIXCONCSMCFG" on page 3-49		X	None	
"INFORMIXCONRETRY" on page 3-49	X	X	None	JDBC
"INFORMIXCONTIME" on page 3-50	X	X	None	JDBC
"INFORMIXCPPMAP" on page 3-51		X	None	
"INFORMIXDIR" on page 3-51	X	X	None	
"INFORMIXOPCACHE" on page 3-52		X	Optical Subsystem only	JDBC
"INFORMIXSERVER" on page 3-52	X	X	None	JDBC
INFORMIXSERVER_SECONDARY				JDBC
"INFORMIXSHMBASE (UNIX)" on page 3-53	X	X	UNIX only	
"INFORMIXSQLHOSTS" on page 3-53	X	X	None	
"INFORMIXSTACKSIZE" on page 3-54	X	X	None	JDBC
"INFORMIXTERM Environment Variable (UNIX)" on page 3-54	X	X	DB-Access, UNIX only	
"INF_ROLE_SEP" on page 3-55		X	None	

Table 3-1. Alphabetical list of environment variables (continued)

Environment variable	XPS	Informix	Restrictions	See
"INTERACTIVE_DESKTOP_OFF (Windows)" on page 3-55		X	Windows only	
"ISM_COMPRESSION" on page 3-56	X	X	ISM, ON–Bar only	
"ISM_DEBUG_FILE" on page 3-56	X	X	ISM only	
"ISM_DEBUG_LEVEL" on page 3-56	X	X	ISM, ON–Bar only	
"ISM_ENCRYPTION" on page 3-57	X	X	ISM, ON–Bar only	
"ISM_MAXLOGSIZE" on page 3-57	X	X	ISM only	
"ISM_MAXLOGVERS" on page 3-57	X	X	ISM only	
"JAR_TEMP_PATH" on page 3-58		X	JVM	
"JAVA_COMPILER" on page 3-58		X	JVM	
JDBCTEMP				JDBC
"JVM_MAX_HEAP_SIZE" on page 3-58		X	JVM	
"LD_LIBRARY_PATH (UNIX)" on page 3-58			SQL APIs, UNIX only	
"LIBERAL_MATCH (XPS)" on page 3-59	X		None	
"LIBPATH (UNIX)" on page 3-59			SQL APIs, UNIX only	
LOBCACHE				JDBC
LOGINTIMOUT				JDBC
NEWNLSMAP				JDBC
"NODEFDAC" on page 3-60	X	X	None	JDBC
"ONCONFIG" on page 3-60	X	X	None	
"ONCONFIG" on page 3-60				
"ONINIT_STDOUT (Windows)" on page 3-61		X	Windows only	
"OPTCOMPIND environment variable" on page 3-61	X	X	None	JDBC
"OPTMSG" on page 3-62			ESQL/C only	
"OPTOFC environment variable" on page 3-62			ESQL/C only	JDBC
"OPT_GOAL (Informix, UNIX)" on page 3-63		X	UNIX only	JDBC
"PATH" on page 3-63	X	X	None	JDBC
"PDQPRIORITY" on page 3-64	X	X	None	JDBC
"PLCONFIG environment variable" on page 3-65		X	HPL only	
"PLOAD_LO_PATH" on page 3-65		X	HPL only	JDBC
"PLOAD_SHMBASE" on page 3-66		X	HPL only	
PORTNO_SECONDARY				JDBC
"PSORT_DBTEMP environment variable" on page 3-66	X	X	None	JDBC
"PSORT_NPROCS" on page 3-67	X	X	None	JDBC
PROXY				JDBC
"RTREE_COST_ADJUST_VALUE" on page 3-68		X	None	
SECURITY				JDBC
SERVER_LOCALE	X	X	None	GLS
"SHLIB_PATH (UNIX)" on page 3-68	X	X	UNIX only	

Table 3-1. Alphabetical list of environment variables (continued)

Environment variable	XPS	Informix	Restrictions	See
SQLH_TYPE				JDBC
SQLIDEBUG				JDBC
"STMT_CACHE" on page 3-68		X	None	JDBC
"TERM (UNIX)" on page 3-69	X	X	UNIX only	
"TERMCAP Environment Variable (UNIX)" on page 3-69	X	X	UNIX only	
"TERMINFO Environment Variable (UNIX)" on page 3-70	X	X	UNIX only	
"THREADLIB (UNIX)" on page 3-70			Informix ESQL/C, UNIX only	
"TOBIGINT (XPS)" on page 3-70	X		**dbschema** only	
USE_DTENV		X	None	*IBM Informix ESQL/C Programmer's Manual*
"USETABLENAME" on page 3-71		X	None	
USEV5SERVER				JDBC
"XFER_CONFIG (XPS)" on page 3-71	X		None	

Tip: You might encounter references to environment variables that are not listed in this table. Most likely, these environment variables are not supported in this release or are used to maintain compatibility with earlier product versions. For information, see an earlier version of your IBM Informix documentation.

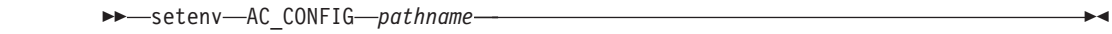
Environment Variables

Sections 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 "Environment Settings" on page 3-6.

AC_CONFIG

You can set the **AC_CONFIG** environment variable to specify the path for the **ac_config.std** configuration file for the **archecker** utility, which checks the validity and completeness of an ON–Bar storage-space backup. The **ac_config.std** file contains default **archecker** configuration parameters.

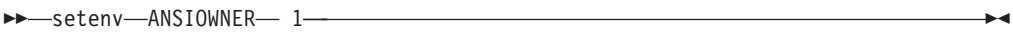


pathname
is the location of the **ac_config.std** configuration file in **$INFORMIXDIR/etc** or **%INFORMIXDIR%\etc**.

For information about **archecker**, see your *IBM Informix Backup and Restore Guide*.

ANSIOWNER

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.



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

Set the **CPFIRST** environment variable to specify the default compilation order for all Informix ESQL/C source files in your programming environment.

When you compile an Informix ESQL/C program 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 an 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
```

DBACCNOIGN

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 nonmenu 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.

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

You can set **DBACCNOIGN** 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 forty-nine rows.

DBANSIWARN

Setting the **DBANSIWARN** environment variable indicates 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.

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

The **DBBLOBBUF** environment variable controls 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.

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

To avoid problems in expanding abbreviated years, applications should require entry of 4-digit years, and should always display years as four digits. The **DBCENTURY** environment variable specifies how to expand literal DATE and DATETIME values that are entered with abbreviated year values.

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.

Examples of Expanding Year Values

The following examples 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. Re-create 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.

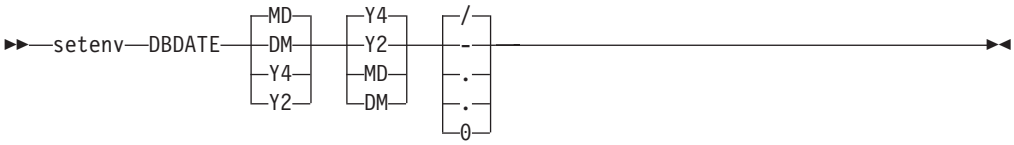
Administration of a database that includes a mix of legacy objects and new objects might become difficult because of differences between the new and the old behavior for evaluating date expressions. To avoid this, it is recommended that you redefine any legacy objects.

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 *and* Extended Parallel Server is not compatible with earlier versions.

DBDATE

The **DBDATE** environment variable specifies the end-user formats of DATE values. On UNIX systems that use the C shell, set **DBDATE** with this syntax.



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. For details of era-based formats, see *IBM Informix GLS User's Guide*.

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, `01/08/2002`). 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, 2005:

DBDATE Setting	Representation of January 8, 2005:		DBDATE Setting	Representation of January 8, 2005:
MDY4/	01/08/2005		Y2DM.	05.08.01
DMY2-	08-01-05		MDY20	010805
MDY4	01/08/2005		Y4MD*	2005/01/08

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" on page 3-29 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'
```

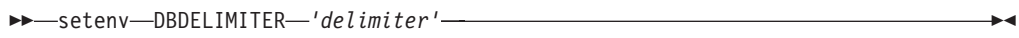
The date of the preceding expression is formatted according to the value of **DBDATE** when the constraint is defined. The check constraint expression is interpreted as `orderdate < '06/25/98'` regardless of the value of **DBDATE** during inserts or updates on the **orderdate** column. Suppose **DBDATE** 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 3-18 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 *and* Extended Parallel Server is not compatible with earlier versions.

DBDELIMITER

The **DBDELIMITER** environment variable specifies the field delimiter used with the **dbexport** utility and with the LOAD and UNLOAD statements.



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

The **DBEDIT** environment variable specifies 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.

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

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.

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

The **DBLANG** environment variable specifies the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the compiled message files that an IBM Informix product uses.

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.

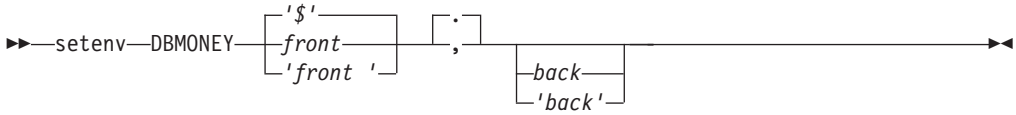
IBM Informix products that use the default U.S. English locale search for message files in the following order:

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

The **DBMONEY** environment variable specifies 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

The **DBONPLOAD** environment variable specifies 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**.

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

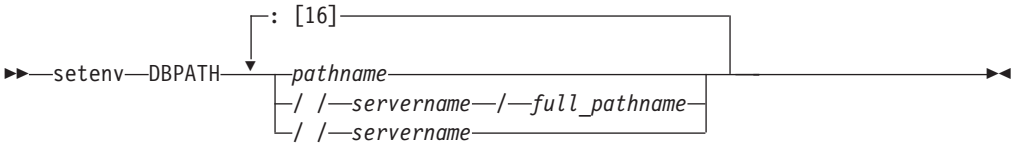
The **DBPATH** environment variable identifies database servers that contain databases. DBPATH can also specify a list of directories (in addition to the current directory) in which DB–Access looks for command scripts (**.sql** files).

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.



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 an IBM Informix 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" on page 3-52. 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

The **DBPRINT** environment variable specifies the default printing program.

program
is 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 (UNIX)

Set 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 pathname.

command
is a command to override the default remote shell.

pathname
is a pathname to override the default remote shell.

If you do not specify the full pathname, the database server searches your **PATH** for the specified *command*. It is highly recommended that you use the full pathname 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 pathname:

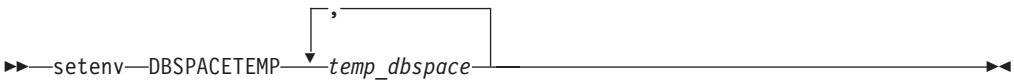
```
setenv DBREMOTECMD /usr/bin/remsh
```

For more information about **DBREMOTECMD**, see the discussion in your *IBM Informix Backup and Restore Guide* about how to use remote tape devices with your database server for archives, restores, and logical-log backups.

DBSPACETEMP

The **DBSPACETEMP** environment variable specifies the dbspaces in which temporary tables are built.

You can list dbspaces, separated by colon (:) or comma (,) symbols to spread temporary space across any number of disks.



temp_dbspace
is the name of a valid existing temporary dbspace.

DBSPACETEMP overrides any default dbspaces that the DBSPACETEMP parameter specifies in the configuration file of the database server. For UPDATE STATISTICS, DBSPACETEMP is used only when you specify the option HIGH. You might have better performance if the list of dbspaces in DBSPACETEMP is composed of chunks that are allocated as raw UNIX devices.

For example, the following command to set the **DBSPACETEMP** environment variable specifies three dbspaces for temporary tables:

```
setenv DBSPACETEMP sorttmp1:sorttmp2:sorttmp3
```

Separate the dbspace entries with either colons or commas. 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. What the IN or FRAGMENT BY clause of a DDL or DML statement specifies
2. For Extended Parallel Server, what a SET TEMP TABLE_SPACE statement specifies
3. On UNIX platforms, the operating-system directory or directories that the environment variable **PSORT_DBTEMP** specifies, if this is set
4. The dbspace or dbspaces that the environment variable **DBSPACETEMP** specifies, if this is set
5. The dbspace or dbspaces that the ONCONFIG parameter DBSPACETEMP specifies.
6. The operating-system file space in **/tmp** (UNIX) or **%temp%** (Windows)
7. For Extended Parallel Server, in a non-critical space, if none of the above are specified
8. For IBM Informix, in the space where the database was created, if none of the above are specified

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, not to 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

The **DBTEMP** environment variable is used by DB-Access and 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.

pathname
is the full pathname 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 pathname **usr/magda/mytemp** for UNIX systems that use the C shell:

```
setenv DBTEMP usr/magda/mytemp
```

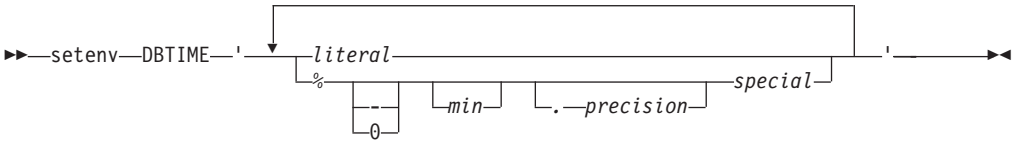
Important: DBTEMP can point to an NFS-mounted directory only if the vendor of that NFS device is certified by IBM.

If **DBTEMP** is not set, the database server creates temporary files in the **/tmp** directory and temporary tables in the DBSPACETEMP directory. See "DBSPACETEMP" on page 3-27 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

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.



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 ≤ *n* ≤ 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 % in the format string).

For example, consider this display format for DATETIME YEAR TO SECOND:

```
Mar 21, 2001 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.

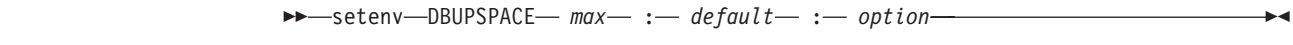
The `F` placeholder does not support this field-width and precision syntax.

Like **DBDATE**, **GL_DATE**, or **GL_DATETIME**, the **DBTIME** setting controls only the character-string representation of data values; it cannot change the internal storage format of the DATETIME column. (For information about formatting DATE values, see the discussion of **DBDATE** on page "DBDATE" on page 3-19.)

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** and **GL_DATE** environment variables.

DBUPSPACE

The **DBUPSPACE** environment variable lets you specify and constrain the amount of system disk space that the UPDATE STATISTICS statement can use when trying to simultaneously construct multiple column distributions.



max is a positive integer, specifying the maximum disk space (in KB) to allocate for sorting in UPDATE STATISTICS operations.

default is a positive integer, specifying the maximum amount of memory (from 4 to 50 megabytes) to allocate without using PDQ.

option An unsigned integer:

- 1: Do not use any indexes for sorting. Print the entire plan for update statistics in sqexplain.out.
- 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 sqexplain.out.

For example, to set **DBUPSPACE** to 2,500 KB of disk space and 1 megabyte of memory, enter this command:

```
setenv DBUPSPACE 2500:1
```

After you set this value, the database server can use no more than 2,500 KB of disk space during the execution of an UPDATE STATISTICS 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.

If you do not set **DBUPSPACE**, the default is one megabyte (1,024 KB) for *max*, and 15 megabytes for *default*. If you attempt to set **DBUPSPACE** to any value less than 1,024 KB, it is automatically set to 1,024 KB, but no error message is returned. If this 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.

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.)

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 types include

- R-tree indexes,
- functional 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).

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 runtime.

Warning: 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 (UNIX)

The **ENVIGNORE** environment variable can deactivate specified environment variable settings in the common (shared) and private environment-configuration files, **informix.rc** and **.informix** respectively.

variable
is 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 user's home directory as **.informix**.

For information about creating or modifying an environment-configuration file, see "Setting Environment Variables in a Configuration File" on page 3-3.

ENVIGNORE itself cannot be set in an environment-configuration file.

FET_BUF_SIZE

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.

size is a positive integer that is larger than the default buffer size, but no greater than 32,767, 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 that is out of the range of SMALLINT values. 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.

GLOBAL_DETACH_INFORM (XPS)

All indexes in IBM Informix Extended Parallel Server are detached. An XPS index is *locally detached* when every index fragment is located in on the same coserver as its associated data fragment and XPS index is *globally detached* if it is fragmented, with index items and their associated data rows located on different coservers.

You should avoid using globally detached indexes because they are inherently less efficient than locally detached indexes.

The **GLOBAL_DETACH_INFORM** environment variable triggers an alarm if a globally detached index is created. The alarm has a severity of 3 (Attention), a Class ID of 10 (Performance Improvement Possible) and a Tag ID of 1 (Globally Detached Index Built).

To enable this alarm, set **GLOBAL_DETACH_INFORM** to any value before starting the server.

Alternatively, you can turn this variable on or off with the **onutil** SET command, as in the following example:

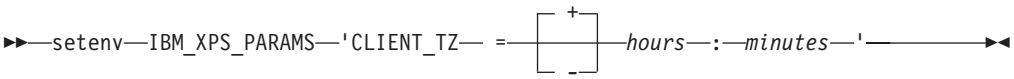
```
% onutil
1> SET GLOBAL_DETACH_INFORM 1;
Dynamic Configuration completed successfully
```

Because **GLOBAL_DETACH_INFORM** is an environment variable and not a configuration parameter, however, it cannot be made persistent with the **onutil** command. Add the variable to an environment-configuration file to avoid setting it each time the server is restarted.

IBM_XPS_PARAMS (XPS)

By default, the CURRENT and TODAY functions return values from the system clock-calendar, based on the location of the server. Use the **IBM_XPS_PARAMS** environment variable to specify a non-default time zone, as an offset from Greenwich Mean Time (GMT), for values returned by CURRENT and TODAY.

To specify an offset from GMT for the built-in CURRENT and TODAY functions, set **IBM_XPS_PARAMS** on the client system to a value before you start the client application, using the following syntax.



hours is an integer in the range from 0 to 13 inclusive, specifying the absolute number of hours in the offset from GMT.

minutes
is a 2-digit integer in the range from 00 to 59 inclusive, specifying any additional minutes in the offset from GMT.

A minus (-) sign before the *hours* specifies a time zone east of GMT, and a positive (*) sign, which is the default, specifies a time zone west of GMT.

This example specifies that CURRENT and TODAY return GMT values:

```
% setenv IBM_XPS_PARAMS 'CLIENT_TZ = 00:00'
```

The next example specifies the Atlantic time zone of eastern Canada:

```
% setenv IBM_XPS_PARAMS 'CLIENT_TZ = +4:00'
```

The **onstat -g ses** command can display the current offset from GMT.

The SET ENVIRONMENT CLIENT_TZ statement of SQL can override the **IBM_XPS_PARAMS** setting, but the default scope of this environment variable is all sessions, rather than only the session in which the SET ENVIRONMENT CLIENT_TZ statement is issued. Reset the GMT offset with the SQL statement for the current session if your application requires a different time zone.

IFMX_CART_ALRM (XPS)

The **IFMX_CART_ALRM** environment variable triggers an alarm if a query executes a Cartesian join. The alarm has a severity of 3 (Attention), a Class ID of 10 (Performance Improvement Possible) and a Tag ID of 2 (Cartesian Join Processing). The alarm message indicates the ID of the session executing the Cartesian join.

To enable this alarm, set **IFMX_CART_ALRM** to any value before starting the database server.

Alternatively, you can turn this variable on or off with the **onutil** SET command:

```
% onutil
1> SET IFMX_CART_ALRM 1;
Dynamic Configuration completed successfully
```

Because **IFMX_CART_ALRM** is an environment variable and not a configuration parameter, it cannot be made persistent by the **onutil** command. Add the variable to an environment-configuration file to avoid setting it each time the server is restarted.

IFMX_HISTORY_SIZE (XPS)

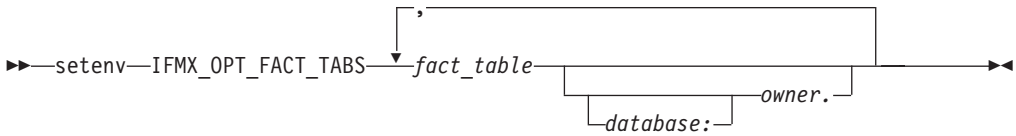
The **IFMX_HISTORY_SIZE** environment variable determines the number of SQL commands that are logged in the DB-Access command history.

value the number of commands stored in the DB-Access history

The default value is 10. The maximum is 100. If a value greater or lower is specified, the default value is used. For more information about using the DB-Access history command, see the *IBM Informix DB-Access User's Guide*.

IFMX_OPT_FACT_TABS (XPS)

The **IFMX_OPT_FACT_TABS** environment variable specifies a list of fact tables that should be used in push-down hash joins whenever possible.



database
is name of the database.

fact_table
is name of the fact table.

owner is name of the table owner.

If you do not specify a database name or owner, the fact table can be in any database or belong to any owner.

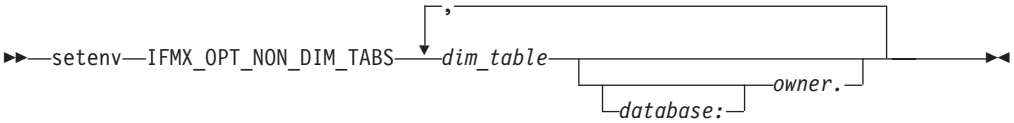
The environment variable lists fact tables for which you want to encourage the optimizer to use push-down hash-join plans. If you do not specify the database name or owner, the table can be in any database or belong to any owner.

When this environment variable is set, push-down hash-join restrictions for the specified fact tables are relaxed to allow the optimizer to use a push-down plan even when the fact table is not larger than the dimension table or when the dimension-table join columns are not unique.

You can use **IFMX_OPT_FACT_TABS** alone to increase the possibility of push-down hash joins. You can also use it in conjunction with the **IFMX_OPT_NON_DIM_TABS** environment variable to fine-tune the use of push-down hash joins.

IFMX_OPT_NON_DIM_TABS (XPS)

The **IFMX_OPT_NON_DIM_TABS** environment variable specifies a list of dimension tables that *cannot* be used in push-down hash-join query plans. If the optimizer detects a fact-dimension table query that joins one of these dimension tables, it does not use a push-down hash-join plan.



database
is name of a database.

dim_table
is name of a dimension table.

owner is name of table owner.

If the database name or owner is not specified, the table can be in any database or can belong to any owner.

When this environment variable is set, if a query joins one of the dimension tables in this list with any fact table, the optimizer never selects a push-down hash join for the query, even if the fact table is included in the **IFMX_OPT_FACT_TABS** list.

You can use the **IFMX_OPT_NON_DIM_TABS** environment variable alone to decrease the possibility of push-down hash joins. You can also use it in conjunction with the **IFMX_OPT_FACT_TABS** environment variable to fine-tune the use of push-down hash joins.

IFX_DEF_TABLE_LOCKMODE

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:

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

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.

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

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. The **IFX_EXTDIRECTIVES** environment variable is set on the client.

You can specify either ON and OFF or 1 and 0 to set the environment variable.

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
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.

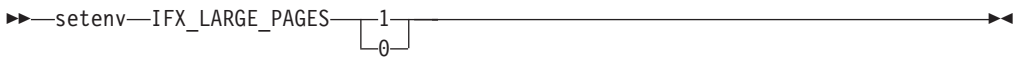
For information about how to define external optimizer directives, see the description of the SAVE EXTERNAL DIRECTIVES statement of SQL in the *IBM Informix Guide to SQL: Syntax*. For more information about the EXT_DIRECTIVES configuration parameter, see the *IBM Informix Administrator's Reference*. For more information about the performance impact of directives, see your *IBM Informix Performance Guide*.

IFX_LARGE_PAGES

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, 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 and Solaris 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.



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 systems.

The DBSA must use operating system commands to configure the large pages. See the operating system documentation for the configuration procedures.

IBM 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.

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 reference

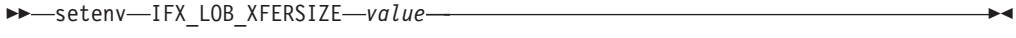
RESIDENT configuration parameter (Administrator's Reference)

IFX_LOB_XFERSIZE

The **IFX_LOB_XFERSIZE** environment variable is used to specify 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. 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 is sent. If **IFX_LOB_XFERSIZE** is not set, the error check occurs after the entire BLOB or CLOB 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.



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

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.

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.

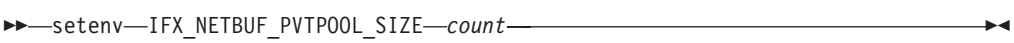
The **IFX_LONGID** setting overrides the internal version of the client application. If the client cannot handle long identifiers despite a newer version number, set **IFX_LONGID** to 0. If the client version can handle long identifiers despite an older version number, set **IFX_LONGID** to 1.

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 (UNIX)

The **IFX_NETBUF_PVTPOOL_SIZE** environment variable specifies the maximum size of the free (unused) private network buffer pool for each database server session.



count is 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

The **IFX_NETBUF_SIZE** environment variable lets you configure the network buffers to the optimum size. It specifies the size of all network buffers in the free (unused) global pool and the private network buffer pool for each database server session.

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 (UNIX)

The **IFX_NO_SECURITY_CHECK** environment variable allows user **informix** or **root** to complete operations with a database server instance even when the 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.

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

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.

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

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.

string Any value prevents the database server from running sysdbopen( ) or sysdblcose( ).

IFX_NOT_STRICT_THOUS_SEP

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.

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.

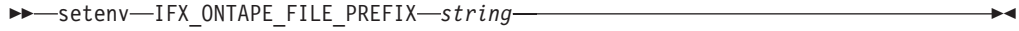
IFX_ONTAPE_FILE_PREFIX

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

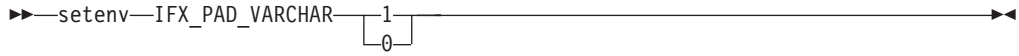
- My_Backup_L1
- My_Backup_L2
- My_Backup_Log0000000001
- My_Backup_Log0000000002



string The prefix to use for the names of backup files.

IFX_PAD_VARCHAR

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.



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

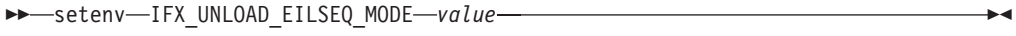
Use the IFX_UNLOAD_EILSEQ_MODE environment variable to help migrate databases from Informix Version 10 to Version 11.50 or 11.70, where character data might be encoded with a codeset that is different than the codeset used to create the Version 10 database.

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 CDSK 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 using the Version 11.50 and 11.70 tools.

+ To use this environment variable, set it to any non-empty string.

+ 
+

+ *value* Any alpha or numeric value. For example: yes, true, or 1. As long as a
+ value is specified.

+ 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 the functionality that is available by setting
+ the EILSEQ_COMPAT_MODE configuration parameter in the ONCONFIG file. The
+ configuration parameter affects character data being inserted into the database.
+ Whereas IFX_UNLOAD_EILSEQ_MODE environment variable affects character
+ data being unloaded from the database.

IFX_UPDDESC

You must set the **IFX_UPDDESC** environment variable at execution time before you can do a DESCRIBE of an UPDATE statement.

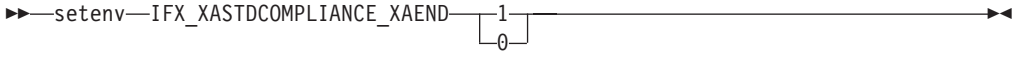
value is any non-NULL value.

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

In earlier releases of IBM Informix, an internal rollback of a global transaction freed the transaction. In releases later than XPS 8.40 and IDS 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.



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

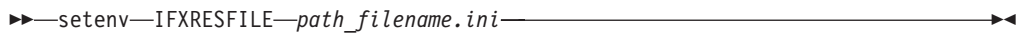
An alternative base address for a utility to attach the server shared memory segments.

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.

IFXRESFILE (Linux)

Set the **IFXRESFILE** environment variable to the path and name of your response file before running an RPM-method installation command. If you want to accept the default IBM Informix installation settings, do not use this environment variable.

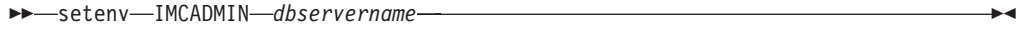


path_filename
specifies the path and name of the response file (.ini file) in which you changed the default installation settings of the **bundle.ini** file shipped with the installation media

For information about creating a response file by customizing the **bundle.ini** file, see the *IBM Informix Installation Guide for UNIX, Linux, and Mac OS X*.

IMCADMIN

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 an IBM Informix product.



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

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.

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

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.

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 (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.

compiler
is the filename of the C compiler.

pathname
is the full pathname 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 using the **-fwritable-strings** option. Failure to do so can produce unpredictable results, possibly including core dumps.

INFORMIXCONCSMCFG

The **INFORMIXCONCSMCFG** environment variable specifies the location of the **concsm.cfg** file that describes communications support modules.

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

The **INFORMIXCONRETRY** environment variable sets the maximum number of *additional* connection attempts that should be made to each database server by the client during the time limit that **INFORMIXCONTIME** specifies.

count is the number of additional attempts to connect to each database server.

For example, the following command sets **INFORMIXCONRETRY** to specify three additional connection attempts (after the initial attempt):

```
setenv INFORMIXCONRETRY 3
```

The default value for **INFORMIXCONRETRY** is one retry after the initial connection attempt. The **INFORMIXCONTIME** setting, described in the following section, takes precedence over the **INFORMIXCONRETRY** setting.

INFORMIXCONTIME

The **INFORMIXCONTIME** environment variable specifies for how many seconds the CONNECT statement continues each attempt 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, but some systems might encounter very 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.

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 attempt to set **INFORMIXCONTIME** to zero, the database server automatically resets it to the default value of 60 seconds.

If execution of the CONNECT statement involves searching **DBPATH**, the following rules apply:

- All appropriate servers in the **DBPATH** setting are accessed at least once, even though the **INFORMIXCONTIME** value might be 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.
- **INFORMIXCONRETRY** specifies how many additional connection attempts should be made for each database server entry in **DBPATH**.

- The **INFORMIXCONTIME** value is divided among the number of database server entries specified in **DBPATH**. Thus, if **DBPATH** contains numerous servers, you should 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.

INFORMIXCONTIME takes precedence over the **INFORMIXCONRETRY** setting. Retry efforts can end after the **INFORMIXCONTIME** value is exceeded, but before the **INFORMIXCONRETRY** value is reached.

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
```

INFORMIXCPPMAP

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.

pathname
is 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

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*.

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
```

INFORMIXOPCACHE

The **INFORMIXOPCACHE** environment variable can specify the size of the memory cache for the staging-area blobspace of the client application.

kilobytes
specifies the value you set for the optical memory cache.

Set the **INFORMIXOPCACHE** environment variable by specifying the size of the memory cache in KB. The specified size must be equal to or smaller than the size of the system-wide configuration parameter, OPCACHEMAX.

If you do not set **INFORMIXOPCACHE**, the default cache size is 128 kilobytes or the size specified in the configuration parameter OPCACHEMAX. The default for OPCACHEMAX is 0. If you set **INFORMIXOPCACHE** to a value of `0`, Optical Subsystem does not use the cache.

INFORMIXSERVER

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 must be set before you can use IBM Informix client products. It has the following syntax.

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.

For Extended Parallel Server, the **INFORMIXSERVER** environment variable specifies the name of a dbserver group. To specify a coserver name, use the following format:

dbservername.coserver_number

Here *dbservername* is the value that you assigned to the DBSERVERNAME configuration parameter in the ONCONFIG configuration file and *coserver_number* is the value that you assigned to the COSERVER configuration parameter for the connection coserver.

Strictly speaking, **INFORMIXSERVER** is not required for initialization. If **INFORMIXSERVER** is not set, however, Extended Parallel Server does not build the **sysmaster** tables.

INFORMIXSHMBASE (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`.

value is an integer (in KB) used to calculate the memory address.

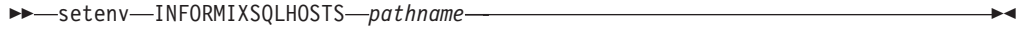
The database server calculates the memory address where segments are attached by multiplying the value of **INFORMIXSHMBASE** by `1,024`. For example, on a system that uses the C shell, you can set the memory address to the value `0x800000` by entering the following command:

```
setenv INFORMIXSHMBASE 8192
```

For more information, see your *IBM Informix Administrator's Guide* and the *IBM Informix Administrator's Reference*.

INFORMIXSQLHOSTS

The **INFORMIXSQLHOSTS** environment variable specifies where the SQL client or the database server can find connectivity information.



pathname
is the full pathname of the connectivity information file.

On UNIX systems, the default search path for the connectivity information file is **$INFORMIXDIR/etc/sqlhosts**.

The following command overrides this default to specify the **mysqlhosts** file in the **/work/envt** directory:

```
setenv INFORMIXSQLHOSTS /work/envt/mysqlhosts
```

On Windows, **INFORMIXSQLHOSTS** points to the computer whose registry contains the SQLHOSTS subkey.

The next example specifies that the client or database server look for connectivity information about a computer named **arizona**:

```
set INFORMIXSQLHOSTS = \\arizona
```

For details of the information that **sqlhosts** (or a file with a non-default filename) can provide about connectivity, see your *IBM Informix Administrator's Guide*.

INFORMIXSTACKSIZE

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.

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 -STACKSIZE 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.

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 page "TERMCAP Environment Variable (UNIX)" on page 3-69.

INF_ROLE_SEP

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.

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:

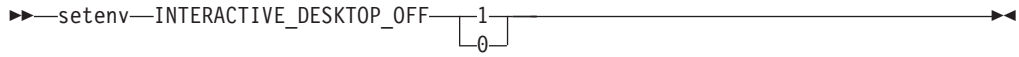
- The Database Server Administrator (DBSA)
- The Audit Analysis Officer (AAO)
- The standard user

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 (Windows)

This environment variable lets you prevent interaction with the Windows desktop when an SPL routine executes a SYSTEM command.



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).

ISM_COMPRESSION

Set this environment variable in the ON–Bar environment to specify whether the IBM Informix Storage Manager should use data compression.

If **ISM_COMPRESSION** is set to TRUE in the environment of the ON–Bar process that makes a request, the ISM server uses a data-compression algorithm to store or retrieve the requested data. If **ISM_COMPRESSION** is set to FALSE or is not set, the ISM server does not use compression.

ISM_DEBUG_FILE

Set the **ISM_DEBUG_FILE** environment variable in the IBM Informix Storage Manager server environment to specify where to write XBSA messages.

pathname
specifies the location of the XBSA message log file.

If you do not set **ISM_DEBUG_FILE**, the XBSA message log is located in the **$INFORMIXDIR/ism/applogs/xbsa.messages** directory on UNIX, or in the **c:\nsr\applogs\xbsa.messages** directory on Windows systems.

ISM_DEBUG_LEVEL

Set the **ISM_DEBUG_LEVEL** environment variable in the ON–Bar environment to control the level of reporting detail recorded in the XBSA messages log. The XBSA shared library writes to this log.

value specifies the level of reporting detail, where 1 ≤ *value* ≤ 9.

If **ISM_DEBUG_LEVEL** is not set, has a null value, or has a value outside this range, the default detail level is 1. A detail level of 0 suppresses all XBSA debugging records. A detail level of 1 reports only XBSA failures.

ISM_ENCRYPTION

Set the **ISM_ENCRYPTION** environment variable in the ON–Bar environment to specify whether IBM Informix Storage Manager uses data encryption.

Three settings of **ISM_ENCRYPTION** are supported:

XOR uses encryption.

NONE does not use encryption.

TRUE uses encryption.

If **ISM_ENCRYPTION** is set to NONE or is not set, the ISM server does not use encryption.

If the **ISM_ENCRYPTION** is set to TRUE or XOR in the environment of the ON–Bar process that makes a request, the ISM server uses encryption to store or retrieve the data specified in that request.

ISM_MAXLOGSIZE

Set the **ISM_MAXLOGSIZE** environment variable in the IBM Informix Storage Manager server environment to specify the size threshold of the ISM activity log.

size specifies the size threshold (in megabytes) of the activity log.

If **ISM_MAXLOGSIZE** is not set, then the default size limit is 1 megabyte. If **ISM_MAXLOGSIZE** is set to a null value, then the threshold is 0 bytes.

ISM_MAXLOGVERS

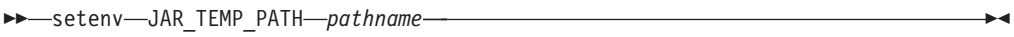
set the **ISM_MAXLOGVERS** environment variable in the IBM Informix Storage Manager server environment to specify the maximum number of activity-log files to be preserved by the ISM server.

value specifies the number of files to be preserved.

If **ISM_MAXLOGVERS** is not set, then the default number of files is four. If the setting is a null value, then the ISM server preserves no activity log files.

JAR_TEMP_PATH

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.



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

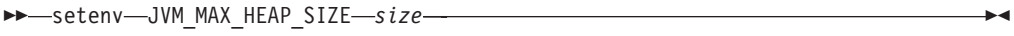
You can set the **JAVA_COMPILER** environment variable in the Java virtual machine environment to disable JIT compilation.

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

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.



size is a positive integer that specifies the maximum size (in megabytes).

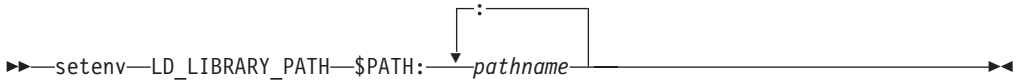
For example, the following command sets the maximum heap size at 12 MB:

```
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 (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
```

LIBERAL_MATCH (XPS)

The **LIBERAL_MATCH** environment variable allows the database server to ignore trailing blanks when the LIKE and MATCHES operators occur in SQL statements that compare two column values.

When this environment variable is set, the database server ignores trailing blanks in a LIKE or MATCHES condition. For example, if **LIBERAL_MATCH** is set, and you specify "`M LIKE P`" when **P** contains trailing blank spaces that do not occur in **M**, the result is `TRUE`. When this environment variable is not set, the database server returns `FALSE` for string comparisons like this that differ only in trailing blank characters.

This environment variable supports behavior consistent with that of the LIKE and MATCHES operators in IBM Informix, Versions 7.x, 9.x, and 10.x. This behavior (like the MATCHES operator) is an extension to the ANSI/ISO standard for SQL.

For more information about the LIKE and MATCHES operators, see the *IBM Informix Guide to SQL: Syntax*.

LIBPATH (UNIX)

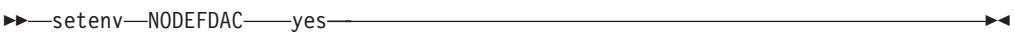
The **LIBPATH** environment variable tells the shell on AIX systems which directories to search for dynamic-link libraries for the INTERSOLV DataDirect ODBC Driver. You must specify the full pathname for the directory where you installed the product.

pathname
specifies the search path for the libraries.

On Solaris, set **LD_LIBRARY_PATH**. On HP-UX, set **SHLIB_PATH**.

NODEFDAC

When the **NODEFDAC** environment variable is set to yes, it prevents default table privileges (Select, Insert, Update, and Delete) from being granted to PUBLIC when a new table is created during the current session in a database that is not ANSI-compliant.



yes prevents default table privileges from being granted to PUBLIC on new tables in a database that is not ANSI-compliant. This setting also prevents the Execute privilege for a new user-defined routine from being granted to PUBLIC by default when the 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.

ONCONFIG

The **ONCONFIG** environment variable specifies the name of the active file that holds configuration parameters for the database server. This file is read as input during the initialization procedure. After you prepare the ONCONFIG configuration file, set **ONCONFIG** to the name of this file.

filename
is the name of a file in **$INFORMIXDIR/etc** that contains the configuration parameters for your database.

To prepare the ONCONFIG file, make a copy of the **onconfig.std** file and modify the copy. It is recommended that you 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.

To prepare the ONCONFIG file for Extended Parallel Server, make a copy of the **onconfig.std** file if you are using a single coserver configuration or make a copy of the **onconfig.xps** file if you are using a multiple coserver configuration. You can use the **onconfig.std** file for a multiple coserver configuration, but you would be required to add additional keywords and configuration parameters such as END, NODE, and COSERVER, which are already provided for you in the **onconfig.xps** file.

If the **ONCONFIG** environment variable is not set, the database server uses configuration values from either the $**ONCONFIG** file or the **$INFORMIXDIR/etc/onconfig** file.

For more information about configuration parameters and the ONCONFIG file, see the *IBM Informix Administrator's Reference*.

ONINIT_STDOUT (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.

The **ONINIT_STDOUT** environment variable is valid only on Windows platforms.

If **ONINIT_STDOUT** is not set, then output generated by the **oninit** command will not be saved.

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.

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

Set the **OPTMSG** environment variable at runtime before you start an IBM Informix ESQL/C application to enable (or disable) optimized message transfers (message chaining) for all SQL statements in an application.

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 an IBM Informix ESQL/C application or other APIs (such as JDBC, ODBC, OLE DB, LIBDMI, and Lib C++) that use DECLARE and OPEN statements to establish a cursor.

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 (Informix, 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.

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

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.

pathname
specifies the search path for the executables.

Include a colon (:) separator between the pathnames on UNIX systems. (Use the semicolon (;) separator between pathnames on Windows systems.)

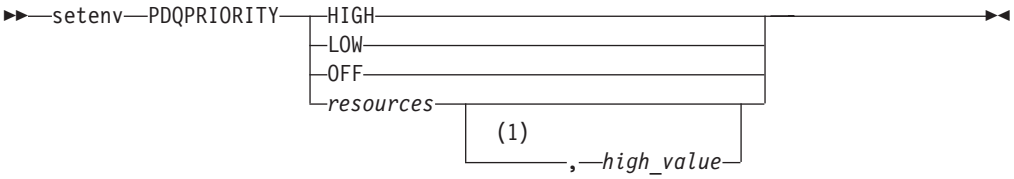
You can specify the search path in various ways. The **PATH** environment variable tells the operating system where to search for executable programs. You must include the directory that contains your IBM Informix product in your **path** setting before you can use the product. This directory should 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

For IBM Informix, 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.

For Extended Parallel Server, the **PDQPRIORITY** environment variable determines only the allocation of memory resources.



Notes:

1 Extended Parallel Server only

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).

high_value

Optional integer value that requests the maximum percentage of memory (for Extended Parallel Server only). When you specify this value after the *resources* value, you request a range of memory, expressed as a percentage.

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 IBM 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.

Using PDQPRIORITY with Extended Parallel Server

The *resources* value establishes the minimum percentage of memory when you also specify *high_value* to request a range of memory allocation. Other parallel operations can occur when the **PDQPRIORITY** setting is LOW.

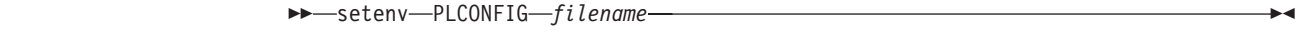
When the **PDQPRIORITY** environment variable is not set, the default value is the value of the PDQPRIORITY configuration parameter.

When **PDQPRIORITY** is set to `0`, Extended Parallel Server can execute a query in parallel, depending on the number of available processors, the fragmentation of tables queried, the complexity of the query, and other factors. **PDQPRIORITY** does not affect the degree of parallelism in Extended Parallel Server.

An application can prevent changes to the **PDQPRIORITY** setting with the SET PDQPRIORITY IMMUTABLE or SET ALL_MUTABLES statements of SQL. You can also override the setting of this environment variable by issuing the SQL statement SET ENVIRONMENT to change the IMPLICIT_PDQ or BOUNT_IMPL_PDQ options, as the *IBM Informix Guide to SQL: Syntax* describes.

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.



filename
specifies the simple filename 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

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).

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

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.

value is 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*.

PSORT_DBTEMP environment variable

The **PSORT_DBTEMP** environment variable specifies the location where the database server writes the temporary files that the environment variable uses to perform a sort.

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" on page 3-27.

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

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.

PSORT_NPROCS does not necessarily improve sorting speed for Extended Parallel Server, because the database server sorts in parallel whether this environment variable is set or not.

Before the sorting package performs a parallel sort, make sure that the database server has enough memory for the sort.

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 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. 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 Values for Attached Indexes

The default number of threads is different for attached indexes.

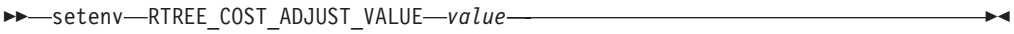
If the **PSORT_NPROCS** environment variable is set, you get the specified number of sort threads for each fragment of the index that is being built.

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

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.

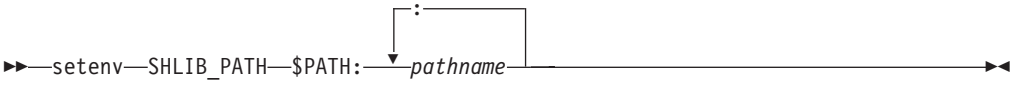


value is a floating-point number, where 1 ≤ *value* ≤ 1000, specifying a multiplier for estimating the cost of using an index on a UDT column.

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 (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.

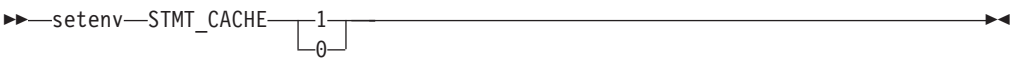


pathname
specifies the search path for the libraries.

On Solaris systems, set **LD_LIBRARY_PATH**. On AIX systems, set **LIBPATH**.

STMT_CACHE

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.



1 enables the SQL statement cache.

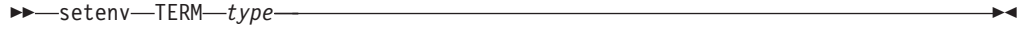
0 disables the SQL statement cache.

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 (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.



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.

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.

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 (UNIX)

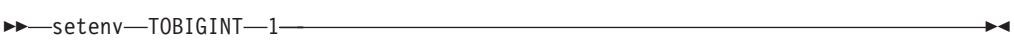
Use the **THREADLIB** environment variable to compile multithreaded Informix ESQL/C applications. A multithreaded Informix ESQL/C application lets you establish as many connections to one or more databases as there are threads. These connections can remain active while the application program executes.

The **THREADLIB** environment variable indicates which thread package to use when you compile an application. Currently only the Distributed Computing Environment (DCE) is supported.

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.

TOBIGINT (XPS)

You can use the **TOBIGINT** environment variable to change the default **INT8** label that the **dbschema** utility displays in its output for columns of the **INT8** data type to the string **BIGINT**.



Set **TOBIGINT** to 1 to enable, and unset **TOBIGINT** to disable this **dbschema** functionality. The name **BIGINT** is the identifier of a built-in 8-byte integer data type of DB2® database servers of IBM. See the Migration Guide for additional information about the **TOBIGINT** environment variable.

USETABLENAME

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.

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.

XFER_CONFIG (XPS)

The **XFER_CONFIG** environment variable specifies the location of the **xfer_config** configuration file.

pathname
specifies the location of the **xfer_config** file.

The **xfer_config** file works with the **onxfer** utility to help users migrate from Version 7.x to Version 8.x. It contains various configuration parameter settings that users can modify and a list of tables that users can select to be transferred.

The default **xfer_config** file is located in the **$INFORMIXDIR/etc** directory on UNIX systems or in the **%INFORMIXDIR%\etc** directory in Windows.

Index of Environment Variables

Table 3-4 on page 3-72 provides an overview of the uses for the various IBM Informix and UNIX environment variables. This serves as an index to general topics and lists the related environment variables and the pages where the environment variables are introduced. Where the **Topic** column is empty, the entry refers to the previously listed topic.

The term *GLS Guide* in the **Page** column in Table 3-4 on page 3-72 indicates environment variables that are described in the *IBM Informix GLS User's Guide*.

The term *ER Guide* in the **Page** column in Table 3-4 indicates environment variables that are described in the *IBM Informix Enterprise Replication Guide*.

Table 3-4. Uses for Environment Variables

Topic		Environment Variable	See
Abbreviated year values		**DBCENTURY**	"DBCENTURY" on page 3-16
Alarms for SQL operations			
	Globally detached indexes	**GLOBAL_DETACH_INFRM**	"GLOBAL_DETACH_INFORM (XPS)" on page 3-35
	Cartesian joins	**IFMX_CART_ALARM**	"IFMX_CART_ALRM (XPS)" on page 3-36
ANSI/ISO SQL compliance			
	Lettercase of owner names	**ANSIOWNER**	"ANSIOWNER" on page 3-14
	IBM Informix syntax extensions	**DBANSIWARN**	"DBANSIWARN" on page 3-16
	default table privileges	**NODEFDAC**	"NODEFDAC" on page 3-60
archecker utility		**AC_CONFIG**	"AC_CONFIG" on page 3-13
Buffer: fetch size		**FET_BUF_SIZE**	"FET_BUF_SIZE" on page 3-34
	network size	**IFX_NETBUF_SIZE**	"IFX_PAD_VARCHAR" on page 3-45
	network pool size	**IFX_NETBUF_PVTPOOL_SIZE**	"IFX_NETBUF_PVTPOOL_SIZE (UNIX)" on page 3-42
	BYTE or TEXT data buffer	**DBBLOBBUF**	"DBBLOBBUF" on page 3-16
Cache: enabling		**STMT_CACHE**	"STMT_CACHE" on page 3-68
	size for Optical Subsystem	**INFORMIXOPCACHE**	"INFORMIXOPCACHE" on page 3-52
Client/server:			
	default server	**INFORMIXSERVER**	"INFORMIXSERVER" on page 3-52
	shared memory segments	**INFORMIXSHMBASE**	"INFORMIXSHMBASE (UNIX)" on page 3-53
	stacksize for client session	**INFORMIXSTACKSIZE**	"INFORMIXSTACKSIZE" on page 3-54
	locale of client, server	**CLIENT_LOCALE** **DB_LOCALE**	GLS Guide
	locale for file I/O	**SERVER_LOCALE**	GLS Guide
Code-set conversion			
	code set of client, server	**CLIENT_LOCALE** **DB_LOCALE**	GLS Guide
	concsm.cfg file	**INFORMIXCONCSMCFG**	"INFORMIXCONCSMCFG" on page 3-49
	Compiler:	**INFORMIXC**	"INFORMIXC (UNIX)" on page 3-49
	multibyte characters	**CC8BITLEVEL** **GL_USEGLU**	GLS Guide
	C++	**INFORMIXCPPMAP**	"INFORMIXCPPMAP" on page 3-51
	ESQL/C	**THREADLIB**	"TOBIGINT (XPS)" on page 3-70
Configuration file:			
	database server	**ONCONFIG**	"ONCONFIG" on page 3-60
	ignore environment variables	**ENVIGNORE**	"ENVIGNORE (UNIX)" on page 3-34

Table 3-4. Uses for Environment Variables (continued)

Topic		Environment Variable	See
	Configuration parameter: COSERVER	**INFORMIXSERVER**	"INFORMIXSERVER" on page 3-52
	DBSERVERNAME	**INFORMIXSERVER**	"INFORMIXSERVER" on page 3-52
	DBSPACETEMP	**DBSPACETEMP**	"DBSPACETEMP" on page 3-27
	DIRECTIVES	**IFX_DIRECTIVES** **IFX_EXTDIRECTIVES**	"IFX_DIRECTIVES" on page 3-39 "IFX_EXTDIRECTIVES" on page 3-39
	OPCACHEMAX	**INFORMIXOPCACHE**	"INFORMIXOPCACHE" on page 3-52
	OPTCOMPIND	**OPTCOMPIND**	"OPTCOMPIND environment variable" on page 3-61
	OPT_GOAL	**OPT_GOAL**	"OPT_GOAL (Informix, UNIX)" on page 3-63
	PDQPRIORITY	**PDQPRIORITY**	"PDQPRIORITY" on page 3-64
	STACKSIZE	**INFORMIXSTACKSIZE**	"INFORMIXSTACKSIZE" on page 3-54
Connecting		**INFORMIXCONRETRY** **INFORMIXCONTIME** **INFORMIXSERVER** **INFORMIXSQLHOSTS**	"INFORMIXCONRETRY" on page 3-49 "INFORMIXCONRETRY" on page 3-49 "INFORMIXSERVER" on page 3-52 "INFORMIXSQLHOSTS" on page 3-53
Data distributions		**DBUPSPACE**	"DBUPSPACE" on page 3-31
Database locale		**DB_LOCALE**	GLS Guide
Database server		**INFORMIXSERVER**	"INFORMIXSERVER" on page 3-52
	locale for file I/O	**SERVER_LOCALE**	GLS Guide
	configuration file	**ONCONFIG**	"ONCONFIG" on page 3-60
	parallel sorting	**PSORT_DBTEMP** **PSORT_NPROCS**	"PSORT_DBTEMP environment variable" on page 3-66 "PSORT_NPROCS" on page 3-67
	parallelism	**PDQPRIORITY**	"PDQPRIORITY" on page 3-64
	role separation	**INF_ROLE_SEP**	"INF_ROLE_SEP" on page 3-55
	shared memory	**INFORMIXSHMBASE**	"INFORMIXSHMBASE (UNIX)" on page 3-53
	stacksize	**INFORMIXSTACKSIZE**	"INFORMIXSTACKSIZE" on page 3-54
	temporary tables	**DBSPACETEMP** **DBTEMP** **PSORT_DBTEMP**	"DBSPACETEMP" on page 3-27 "DBTEMP" on page 3-28 "PSORT_DBTEMP environment variable" on page 3-66
	variable-length packets	**IFX_PAD_VARCHAR**	"IFX_PAD_VARCHAR" on page 3-45
Date and time values, formats		**DBCENTURY** **DBDATE** **GL_DATE** **DBTIME** **GL_DATETIME** **IBM_XPS_PARAMS** **USE_DTENV**	"DBCENTURY" on page 3-16 "DBDATE" on page 3-19: GLS Guide "DBTIME" on page 3-29: GLS Guide "IBM_XPS_PARAMS (XPS)" on page 3-36 *IBM Informix ESQL/C Programmer's Manual*

Table 3-4. Uses for Environment Variables (continued)

Topic	Environment Variable	See
DB-Access utility	**DBANSIWARN** **DBDELIMITER** **DBEDIT** **DBFLTMASK** **DBPATH** **FET_BUF_SIZE** **INFORMIXSERVER** **INFORMIXTERM** **TERM** **TERMCAP** **TERMINFO**	"DBANSIWARN" on page 3-16 "DBDELIMITER" on page 3-21 "DBEDIT" on page 3-21 "DBFLTMASK" on page 3-22 "DBPATH" on page 3-24 "FET_BUF_SIZE" on page 3-34 "INFORMIXSERVER" on page 3-52 "INFORMIXTERM Environment Variable (UNIX)" on page 3-54 "TERM (UNIX)" on page 3-69 "TERMCAP Environment Variable (UNIX)" on page 3-69 "TERMINFO Environment Variable (UNIX)" on page 3-70
dbexport utility	**DBDELIMITER**	"DBDELIMITER" on page 3-21
dbschema utility	**TOBIGINT**	"TOBIGINT (XPS)" on page 3-70
Delimited identifiers	**DELIMIDENT**	"DELIMIDENT environment variable" on page 3-33
Disk space	**DBUPSPACE**	"DBUPSPACE" on page 3-31
Editor	**DBEDIT**	"DBEDIT" on page 3-21
ESQL/C: ANSI compliance	**DBANSIWARN**	"DBANSIWARN" on page 3-16
C compiler	**INFORMIXC**	"INFORMIXC (UNIX)" on page 3-49
DATETIME formatting	**DBTIME**	"DBTIME" on page 3-29; GLS Guide
delimited identifiers	**DELIMIDENT**	"DELIMIDENT environment variable" on page 3-33
multibyte characters	**CLIENT_LOCALE** **ESQLMF** **GL_USEGLU**	GLS Guide
multithreaded applications	**THREADLIB**	"TOBIGINT (XPS)" on page 3-70
C preprocessor	**CPFIRST**	"CPFIRST" on page 3-14
Executable programs	**PATH**	"PATH" on page 3-63
Fetch buffer size	**FET_BUF_SIZE**	"FET_BUF_SIZE" on page 3-34
Filenames: multibyte	**GLS8BITFSYS**	GLS Guide
Files: field delimiter	**DBDELIMITER**	"DBDELIMITER" on page 3-21
Files: installation	**INFORMIXDIR**	"INFORMIXDIR" on page 3-51
Files: locale	**CLIENT_LOCALE** **DB_LOCALE** **SERVER_LOCALE**	GLS Guide
Files: map for C++	**INFORMIXCPPMAP**	"INFORMIXCPPMAP" on page 3-51
Files: message	**DBLANG**	"DBLANG" on page 3-22
Files: temporary	**DBSPACETEMP**	"DBSPACETEMP" on page 3-27
Files: temporary, for Gateways	**DBTEMP**	"DBTEMP" on page 3-28
Files: temporary sorting	**PSORT_DBTEMP**	"PSORT_DBTEMP environment variable" on page 3-66
Files: **termcap**, **terminfo**	**INFORMIXTERM** **TERM** **TERMCAP** **TERMINFO**	"INFORMIXTERM Environment Variable (UNIX)" on page 3-54 "TERM (UNIX)" on page 3-69 "TERMCAP Environment Variable (UNIX)" on page 3-69 "TERMINFO Environment Variable (UNIX)" on page 3-70

Table 3-4. Uses for Environment Variables (continued)

Topic	Environment Variable	See
Formats: date and time	**DBDATE GL_DATE** **DBTIME** **GL_DATETIME**	"DBDATE" on page 3-19; GLS Guide "DBTIME" on page 3-29; GLS Guide
Format: money	**DBMONEY**	"DBMONEY" on page 3-23, GLS Guide
Gateways	**DBTEMP**	"DBTEMP" on page 3-28
High-Performance Loader	**DBONPLOAD** **PLCONFIG** **PLOAD_LO_PATH** **PLOAD_SHMBASE**	"DBONPLOAD" on page 3-24 "PLCONFIG environment variable" on page 3-65 "PLOAD_LO_PATH" on page 3-65 "PLOAD_SHMBASE" on page 3-66
Identifiers: delimited	**DELIMIDENT**	"DELIMIDENT environment variable" on page 3-33
Identifiers: longer than 18 bytes	**IFX_LONGID**	"IFX_LONGID" on page 3-42
Identifiers: multibyte characters	**CLIENT_LOCALE** **ESQLMF**	GLS Guide
IBM Informix Storage Manager	**ISM_COMPRESSION** **ISM_DEBUG_FILE** **ISM_DEBUG_LEVEL** **ISM_ENCRYPTION**	"ISM_COMPRESSION" on page 3-56 "ISM_DEBUG_FILE" on page 3-56 "ISM_DEBUG_LEVEL" on page 3-56 "ISM_ENCRYPTION" on page 3-57
IBM Informix Storage Manager	**ISM_MAXLOGSIZE** **ISM_MAXLOGVERS**	"ISM_MAXLOGSIZE" on page 3-57 "ISM_MAXLOGVERS" on page 3-57
Installation	**INFORMIXDIR** **PATH**	"INFORMIXDIR" on page 3-51 "PATH" on page 3-63
Language environment	**DBLANG** See also "Nondefault Locale"	"DBLANG" on page 3-22, GLS Guide
Libraries	**LD_LIBRARY_PATH** **LIBPATH** **SHLIB_PATH**	"LD_LIBRARY_PATH (UNIX)" on page 3-58 "LIBPATH (UNIX)" on page 3-59 "SHLIB_PATH (UNIX)" on page 3-68
Locale	**CLIENT_LOCALE** **DB_LOCALE** **SERVER_LOCALE**	GLS Guide
Lock Mode	**IFX_DEF_TABLE_LOCKMODE**	"IFX_DEF_TABLE_LOCKMODE" on page 3-38
Long Identifiers	**IFX_LONGID**	"IFX_LONGID" on page 3-42
Map file for C++	**INFORMIXCPPMAP**	"INFORMIXCPPMAP" on page 3-51
Message chaining	**OPTMSG**	"OPTMSG" on page 3-62
Message files	**DBLANG**	"DBLANG" on page 3-22, GLS Guide
Money format	**DBMONEY**	"DBMONEY" on page 3-23, GLS Guide
Multibyte characters	**CLIENT_LOCALE** **DB_LOCALE** **SERVER_LOCALE** **GL_USEGLU**	GLS Guide
Multibyte filter	**ESQLMF**	GLS Guide
Multithreaded applications	**THREADLIB**	"TOBIGINT (XPS)" on page 3-70
Network	**DBPATH**	"DBPATH" on page 3-24
Nondefault locale	**CLIENT_LOCALE** **DB_LOCALE** **SERVER_LOCALE**	GLS Guide

Table 3-4. Uses for Environment Variables (continued)

Topic	Environment Variable	See
ON–Bar utility	**ISM_COMPRESSION** **ISM_DEBUG_LEVEL** **ISM_ENCRYPTION**	"ISM_COMPRESSION" on page 3-56 "ISM_DEBUG_LEVEL" on page 3-56 "ISM_ENCRYPTION" on page 3-57
ONCONFIG parameters	See "Configuration parameter"	
oninit output (Windows only)	**ONINIT_STDOUT**	"ONINIT_STDOUT (Windows)" on page 3-61
Optical Subsystem	**INFORMIXOPCACHE**	"INFORMIXOPCACHE" on page 3-52
Optimization: directives	**IFX_DIRECTIVES** **IFX_EXTDIRECTIVES**	"IFX_DIRECTIVES" on page 3-39 "IFX_EXTDIRECTIVES" on page 3-39
Optimization: message transfers	**OPTMSG**	"OPTMSG" on page 3-62
Optimization: join method	**OPTCOMPIND**	"OPTCOMPIND environment variable" on page 3-61
Optimization: performance goal	**OPT_GOAL**	"OPT_GOAL (Informix, UNIX)" on page 3-63
OPTOFC feature	**OPTOFC**	"OPTOFC environment variable" on page 3-62
Parameters	See "Configuration parameter"	
Pathname: **archecker** config file	**AC_CONFIG**	"AC_CONFIG" on page 3-13
Pathname: C compiler	**INFORMIXC**	"INFORMIXC (UNIX)" on page 3-49
Pathname: database files	**DBPATH**	"DBPATH" on page 3-24
Pathname: executable programs	**PATH**	"PATH" on page 3-63
Pathname: HPL sblob handles	**PLOAD_LO_PATH**	"PLOAD_LO_PATH" on page 3-65
Pathname: installation	**INFORMIXDIR**	"INFORMIXDIR" on page 3-51
Pathname: libraries	**LD_LIBRARY_PATH** **LIBPATH** **SHLIB_PATH**	"LD_LIBRARY_PATH (UNIX)" on page 3-58 "LIBPATH (UNIX)" on page 3-59 "SHLIB_PATH (UNIX)" on page 3-68
Pathname: message files	**DBLANG**	"DBLANG" on page 3-22, GLS Guide
Pathname: parallel sorting	**PSORT_DBTEMP**	"PSORT_DBTEMP environment variable" on page 3-66
Pathname: remote shell	**DBREMOTECMD**	"DBREMOTECMD (UNIX)" on page 3-26
Pathname: **xfer_config** file	**XFER_CONFIG**	"XFER_CONFIG (XPS)" on page 3-71
Preserve owner name lettercase	**ANSIOWNER**	"ANSIOWNER" on page 3-14
Printing	**DBPRINT**	"DBPRINT" on page 3-26
Privileges	**NODEFDAC**	"INF_ROLE_SEP" on page 3-55
Query: optimization	**IFX_DIRECTIVES** **IFX_EXTDIRECTIVES** **IFMX_OPT_FACT_TABS** **IFMX_OPT_NON_DIM_TABS** **OPTCOMPIND** **OPT_GOAL** **RTREE_COST_ADJUST_VALUE**	"IFX_DIRECTIVES" on page 3-39 "IFX_EXTDIRECTIVES" on page 3-39 "IFMX_OPT_FACT_TABS (XPS)" on page 3-37 "IFMX_OPT_NON_DIM_TABS (XPS)" on page 3-38 "OPTCOMPIND environment variable" on page 3-61 "OPT_GOAL (Informix, UNIX)" on page 3-63 "RTREE_COST_ADJUST_VALUE" on page 3-68
Query: prioritization	**PDQPRIORITY**	"PDQPRIORITY" on page 3-64
Remote shell	**DBREMOTECMD**	"DBREMOTECMD (UNIX)" on page 3-26

Table 3-4. Uses for Environment Variables (continued)

Topic		Environment Variable	See
Role separation		**INF_ROLE_SEP**	"INF_ROLE_SEP" on page 3-55
Rolled-back transactions		**DBACCNOIGN** **IFX_XASTDCOMPLIANCE_XAEND**	"DBACCNOIGN" on page 3-15 "IFX_XASTDCOMPLIANCE_XAEND" on page 3-46
Routine: DATETIME formatting		**DBTIME**	"DBTIME" on page 3-29, GLS Guide
Server		See "Database Server"	Database server
Server locale		**SERVER_LOCALE**	GLS Guide
Shared memory		**INFORMIXSHMBASE** **PLOAD_SHMBASE**	"INFORMIXSHMBASE (UNIX)" on page 3-53 "PLOAD_SHMBASE" on page 3-66
Shell: remote		**DBREMOTECMD**	"DBREMOTECMD (UNIX)" on page 3-26
Shell: search path		**PATH**	"PATH" on page 3-63
Sorting		**PSORT_DBTEMP** **PSORT_NPROCS**	"PSORT_DBTEMP environment variable" on page 3-66 "PSORT_NPROCS" on page 3-67
SQL statements:caching		**STMT_CACHE**	"STMT_CACHE" on page 3-68
	CONNECT	**INFORMIXCONTIME** **INFORMIXSERVER**	"INFORMIXCONRETRY" on page 3-49 "INFORMIXSERVER" on page 3-52
	CREATE TEMP TABLE	**DBSPACETEMP**	"DBSPACETEMP" on page 3-27
	DESCRIBE FOR UPDATE	**IFX_UPDDESC**	"IFX_UPDDESC" on page 3-46
	LOAD, UNLOAD	**DBDELIMITER**	"DBDELIMITER" on page 3-21
	LOAD, UNLOAD	**DBBLOBBUF**	"DBBLOBBUF" on page 3-16
	SELECT INTO TEMP	**DBSPACETEMP**	"DBSPACETEMP" on page 3-27
	SET PDQPRIORITY	**PDQPRIORITY**	"PDQPRIORITY" on page 3-64
	SET STMT_CACHE	**STMT_CACHE**	"STMT_CACHE" on page 3-68
	UPDATE STATISTICS	**DBUPSPACE**	"DBUPSPACE" on page 3-31
Stacksize		**INFORMIXSTACKSIZE**	"INFORMIXSTACKSIZE" on page 3-54
String search: trailing blanks		**LIBERAL_MATCH**	"LIBERAL_MATCH (XPS)" on page 3-59
Temporary tables		**DBSPACETEMP** **DBTEMP** **PSORT_DBTEMP**	"DBSPACETEMP" on page 3-27 "DBTEMP" on page 3-28 "PSORT_DBTEMP environment variable" on page 3-66
Terminal handling		**INFORMIXTERM** **TERM** **TERMCAP** **TERMINFO**	"INFORMIXTERM Environment Variable (UNIX)" on page 3-54 "TERM (UNIX)" on page 3-69 "TERMCAP Environment Variable (UNIX)" on page 3-69 "TERMINFO Environment Variable (UNIX)" on page 3-70
Time-limited software license		**IFX_NO_TIMELIMIT_WARNING**	"IFX_NO_TIMELIMIT_WARNING" on page 3-44
Time zone, specifying		**IBM_XPS_PARAMS**	"IBM_XPS_PARAMS (XPS)" on page 3-36

Table 3-4. Uses for Environment Variables (continued)

Topic	Environment Variable	See
Utilities: DB-Access	**DBANSIWARN** **DBDELIMITER** **DBEDIT** **DBFLTMASK** **DBPATH** **FET_BUF_SIZE** **IFMX_HISTORY_SIZER** **INFORMIXSERVER** **INFORMIXTERM** **TERM** **TERMCAP** **TERMINFO**	"DBANSIWARN" on page 3-16 "DBDELIMITER" on page 3-21 "DBEDIT" on page 3-21 "DBFLTMASK" on page 3-22 "DBPATH" on page 3-24 "FET_BUF_SIZE" on page 3-34 "IFMX_HISTORY_SIZE (XPS)" on page 3-37 "INFORMIXSERVER" on page 3-52 "INFORMIXTERM Environment Variable (UNIX)" on page 3-54 "TERM (UNIX)" on page 3-69 "TERMCAP Environment Variable (UNIX)" on page 3-69 "TERMINFO Environment Variable (UNIX)" on page 3-70
Utilities: **dbexport**	**DBDELIMITER**	"DBDELIMITER" on page 3-21
Utilities: ON–Bar	**ISM_COMPRESSION** **ISM_DEBUG_LEVEL** **ISM_ENCRYPTION**	"ISM_COMPRESSION" on page 3-56 "ISM_DEBUG_LEVEL" on page 3-56 "ISM_ENCRYPTION" on page 3-57
Variables: overriding	**ENVIGNORE**	"ENVIGNORE (UNIX)" on page 3-34
Virtual Memory Segments on Large Pages	**IFX_LARGE_PAGES**	"IFX_LARGE_PAGES" on page 3-40
Year values (abbreviated)	**DBCENTURY**	"DBCENTURY" on page 3-16

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.

This appendix contains the following sections:

- The first section describes the structure of the tables in the **stores_demo** database. It identifies the primary key of each table, lists the name and data type of each column, and indicates whether the column has a default value or check constraint. Indexes on columns are also identified and classified as unique, allowing duplicate values.
- The second section ("The stores_demo Database Map" on page A-5) shows a map of the tables in the **stores_demo** database and indicates the relationships among columns.
- The third section ("Primary-Foreign Key Relationships" on page A-5) describes the primary-foreign key relationships among columns in tables.
- The final section ("Data in the stores_demo Database" on page A-9) lists the data contained in each table of the **stores_demo** database.

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*.

Structure of the Tables

The **stores_demo** database contains information about a fictitious sporting-goods distributor that services stores in the western United States. This database includes the following tables:

- **customer** ("The customer Table" on page A-2)
- **orders** ("The orders Table" on page A-2)
- **items** ("The items Table" on page A-2)
- **stock** ("The stock Table" on page A-3)
- **catalog** ("The catalog Table" on page A-3)
- **cust_calls** ("The cust_calls Table" on page A-4)
- **call_type** ("The call_type Table" on page A-4)
- **manufact** ("The manufact Table" on page A-4)
- **state** ("The state Table" on page A-4)

Sections that follow describe each table. The unique identifying value for each table (primary key) is shaded.

The customer Table

The **customer** table contains information about the retail stores that place orders from the distributor. Table A-1 shows the columns of the **customer** table.

The **zipcode** column in Table A-1 is indexed and allows duplicate values.

Table A-1. The customer Table

Column Name	Data Type	Description
customer_num	SERIAL(101)	System-generated customer number
fname	CHAR(15)	First name of store representative
lname	CHAR(15)	Last name of store representative
company	CHAR(20)	Name of store
address1	CHAR(20)	First line of store address
address2	CHAR(20)	Second line of store address
city	CHAR(15)	City
state	CHAR(2)	State (foreign key to **state** table)
zipcode	CHAR(5)	Zipcode
phone	CHAR(18)	Telephone number

The orders Table

The **orders** table contains information about orders placed by the customers of the distributor. Table A-2 shows the columns of the **orders** table.

Table A-2. The orders Table

Column Name	Data Type	Description
order_num	SERIAL(1001)	System-generated order number
order_date	DATE	Date order entered
customer_num	INTEGER	Customer number (foreign key to **customer** table)
ship_instruct	CHAR(40)	Special shipping instructions
backlog	CHAR(1)	Indicates order cannot be filled because the item is backlogged: y = yes n = no
po_num	CHAR(10)	Customer purchase order number
ship_date	DATE	Shipping date
ship_weight	DECIMAL(8,2)	Shipping weight
ship_charge	MONEY(6)	Shipping charge
paid_date	DATE	Date order paid

The items Table

An order can include one or more items. One row exists in the **items** table for each item in an order. Table A-3 on page A-3 shows the columns of the **items** table.

Table A-3. The items Table

Column Name	Data Type	Description
item_num	SMALLINT	Sequentially assigned item number for an order
order_num	INTEGER	Order number (foreign key to **orders** table)
stock_num	SMALLINT	Stock number for item (foreign key to **stock** table)
manu_code	CHAR(3)	Manufacturer code for item ordered (foreign key to **manufact** table)
quantity	SMALLINT	Quantity ordered (value must be > 1)
total_price	MONEY(8)	Quantity ordered * unit price = total price of item

The stock Table

The distributor carries 41 types of sporting goods from various manufacturers. More than one manufacturer can supply an item. For example, the distributor offers racing goggles from two manufacturers and running shoes from six manufacturers.

The **stock** table is a catalog of the items sold by the distributor. Table A-4 shows the columns of the **stock** table.

Table A-4. The stock Table

Column Name	Data Type	Description
stock_num	SMALLINT	Stock number that identifies type of item
manu_code	CHAR(3)	Manufacturer code (foreign key to **manufact** table)
description	CHAR(15)	Description of item
unit_price	MONEY(6,2)	Unit price
unit	CHAR(4)	Unit by which item is ordered: • Each • Pair • Case • Box
unit_descr	CHAR(15)	Description of unit

The catalog Table

The **catalog** table describes each item in stock. Retail stores use this table when placing orders with the distributor. Table A-5 shows the columns of the **catalog** table.

Table A-5. The catalog Table

Column Name	Data Type	Description
catalog_num	SERIAL(10001)	System-generated catalog number
stock_num	SMALLINT	Distributor stock number (foreign key to **stock** table)
manu_code	CHAR(3)	Manufacturer code (foreign key to **manufact** table)
cat_descr	TEXT	Description of item
cat_picture	BYTE	Picture of item (binary data)
cat_advert	VARCHAR(255, 65)	Tag line underneath picture

The cust_calls Table

All customer calls for information about orders, shipments, or complaints are logged. The **cust_calls** table contains information about these types of customer calls. Table A-6 shows the columns of the **cust_calls** table.

Table A-6. The cust_calls Table

Column Name	Data Type	Description
customer_num	INTEGER	Customer number (foreign key to **customer** table)
call_dtime	DATETIME YEAR TO MINUTE	Date and time when call was received
user_id	CHAR(18)	Name of person logging call (default is user login name)
call_code	CHAR(1)	Type of call (foreign key to **call_type** table)
call_descr	CHAR(240)	Description of call
res_dtime	DATETIME YEAR TO MINUTE	Date and time when call was resolved
res_descr	CHAR(240)	Description of how call was resolved

The call_type Table

The call codes associated with customer calls are stored in the **call_type** table. Table A-7 shows the columns of the **call_type** table.

Table A-7. The call_type Table

Column Name	Data Type	Description
call_code	CHAR(1)	Call code
code_descr	CHAR (30)	Description of call type

The manufact Table

Information about the nine manufacturers whose sporting goods are handled by the distributor is stored in the **manufact** table. Table A-8 shows the columns of the **manufact** table.

Table A-8. The manufact Table

Column Name	Data Type	Description
manu_code	CHAR(3)	Manufacturer code
manu_name	CHAR(15)	Name of manufacturer
lead_time	INTERVAL DAY(3) TO DAY	Lead time for shipment of orders

The state Table

The **state** table contains the names and postal abbreviations for the 50 states of the United States. Table A-9 shows the columns of the **state** table.

Table A-9. The state Table

Column Name	Data Type	Description
code	CHAR(2)	State code

Table A-9. The state Table (continued)

Column Name	Data Type	Description
sname	CHAR(15)	State name

The stores_demo Database Map

Figure A-1 displays the joins in the **stores_demo** database. The gray 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.

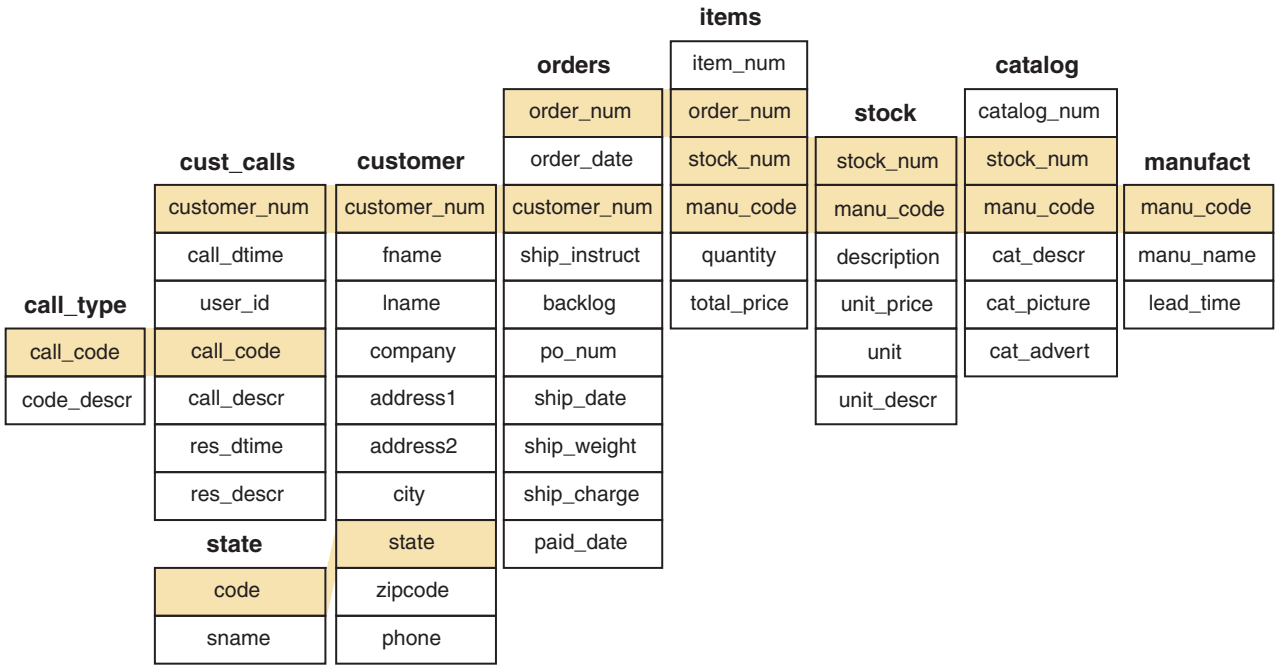


Figure A-1. Joins in the stores_demo Database

Primary-Foreign Key Relationships

The tables of the **stores_demo** database are linked by the primary-foreign key relationships that Figure A-1 shows and are identified in this section. This type of relationship is called a *referential constraint* because a foreign key in one table *references* the primary key in another table. Figure A-2 on page A-6 through Figure A-9 on page A-9 show the relationships among tables and how information stored in one table supplements information stored in others.

The customer and orders Tables

The **customer** table contains a **customer_num** column that holds a number that identifies a customer and columns for the customer name, company, address, and telephone number. For example, the row with information about Anthony Higgins contains the number 104 in the **customer_num** column. The **orders** table also contains a **customer_num** column that stores the number of the customer who placed a particular order. In the **orders** table, the **customer_num** column is a foreign key that references the **customer_num** column in the **customer** table.

Figure A-2 shows this relationship.

customer Table (detail)

customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins

Orders Table (detail)

order_num	order_date	customer_num
1001	05/20/1998	104
1002	05/21/1998	101
1003	05/22/1998	104
1004	05/22/1998	106

Figure A-2. Tables That the customer_num Column Joins

According to Figure A-2, customer 104 (Anthony Higgins) has placed two orders, as his customer number is in two rows of the **orders** table. Because the customer number is a foreign key in the **orders** table, you can retrieve Anthony Higgins's name, address, and information about his orders at the same time.

The orders and items Tables

The **orders** and **items** tables are linked by an **order_num** column that contains an identification number for each order. If an order includes several items, the same order number is in several rows of the **items** table. In the **items** table, the **order_num** column is a foreign key that references the **order_num** column in the **orders** table. Figure A-3 shows this relationship.

orders Table (detail)

order_num	order_date	customer_num
1001	05/20/1998	104
1002	05/21/1998	101
1003	05/22/1998	104

items Table (detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
4	1002	4	HSK
3	1002	3	HSK
9	1003	9	ANZ
8	1003	8	ANZ
5	1003	5	ANZ

Figure A-3. Tables That the order_num Column Joins

The items and stock Tables

The **items** table and the **stock** table are joined by two columns: the **stock_num** column, which stores a stock number for an item, and the **manu_code** column, which stores a code that identifies the manufacturer. You need both the stock number and the manufacturer code to uniquely identify an item. For example, the

item with the stock number 1 and the manufacturer code HRO is a Hero baseball glove; the item with the stock number 1 and the manufacturer code HSK is a Husky baseball glove.

The same stock number and manufacturer code can be in more than one row of the **items** table, if the same item belongs to separate orders. In the **items** table, the **stock_num** and **manu_code** columns are foreign keys that reference the **stock_num** and **manu_code** columns in the **stock** table. Figure A-4 shows this relationship.

items Table (detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ
1	1004	1	HRO

stock Table (detail)

stock_num	manu_code	Description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

Figure A-4. Tables That the stock_num and manu_code Columns Join

The stock and catalog Tables

The **stock** table and **catalog** table are joined by two columns: the **stock_num** column, which stores a stock number for an item, and the **manu_code** column, which stores a code that identifies the manufacturer. You need both columns to uniquely identify an item. In the **catalog** table, the **stock_num** and **manu_code** columns are foreign keys that reference the **stock_num** and **manu_code** columns in the **stock** table. Figure A-5 shows this relationship.

stock Table (detail)

stock_num	manu_code	Description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

catalog Table (detail)

catalog_num	stock_num	manu_code
10001	1	HRO
10002	1	HSK
10003	1	SMT
10004	2	HRO

Figure A-5. Tables That the stock_num and manu_code Columns Join

The stock and manufact Tables

The **stock** table and the **manufact** table are joined by the **manu_code** column. The same manufacturer code can be in more than one row of the **stock** table if the manufacturer produces more than one piece of equipment. In the **stock** table, the **manu_code** column is a foreign key that references the **manu_code** column in the **manufact** table. Figure A-6 shows this relationship.

stock Table (detail)

stock_num	manu_code	Description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

manufact Table (detail)

manu_code	manu_name
NRG	Norge
HSK	Husky
HRO	Hero

Figure A-6. Tables That the manu_code Column Joins

The cust_calls and customer Tables

The **cust_calls** table and the **customer** table are joined by the **customer_num** column. The same customer number can be in more than one row of the **cust_calls** table if the customer calls the distributor more than once with a problem or question. In the **cust_calls** table, the **customer_num** column is a foreign key that references the **customer_num** column in the **customer** table. Figure A-7 shows this relationship.

customer Table (detail)

customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins
105	Raymond	Vector
106	George	Watson

cust_calls Table (detail)

customer_num	call_dtime	user_id
106	1998-06-12 08:20	maryj
127	1998-07-31 14:30	maryj
116	1997-11-28 13:34	mannyh
116	1997-12-21 11:24	mannyh

Figure A-7. Tables That the customer_num Column Joins

The call_type and cust_calls Tables

The **call_type** and **cust_calls** tables are joined by the **call_code** column. The same call code can be in more than one row of the **cust_calls** table because many customers can have the same *type* of problem. In the **cust_calls** table, the **call_code** column is a foreign key that references the **call_code** column in the **call_type** table.

Figure A-8 shows this relationship.

call_type Table (detail)

call_code	code_descr
B	Billing error
D	Damaged goods
I	Incorrect merchandise sent
L	Late shipment
O	Other

cust_calls Table (detail)

customer_num	call_dtime	call_code
106	1998-06-12 08:20	D
127	1998-07-31 14:30	I
116	1997-11-28 13:34	I
116	1997-12-21 11:24	I

Figure A-8. Tables That the call_code Column Joins

The state and customer Tables

The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer** table. If several customers live in the same state, the same state code will be in several rows of the table. In the **customer** table, the **state** column is a foreign key that references the **code** column in the **state** table. Figure A-9 shows this relationship.

customer Table (detail)

customer_num	fname	lname	---	state
101	Ludwig	Pauli	---	CA
102	Carole	Sadler	---	CA
103	Philip	Currie	---	CA

state Table (detail)

code	sname
AK	Alaska
AL	Alabama
AR	Arkansas
AZ	Arizona
CA	California

Figure A-9. Relationship Between the state Column and the code Column

Data in the stores_demo Database

The following tables display the data in the **stores_demo** database.

customer Table

customer_num	fname	lname	company	address1	address2	city	state	zip code	phone
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court		Sunnyvale	CA	94086	408-789-8075

customer_num	fname	lname	company	address1	address2	city	state	zip code	phone
102	Carole	Sadler	Sports Spot	785 Geary Street		San Francisco	CA	94117	415-822-1289
103	Philip	Currie	Phil's Sports	654 Poplar	P. O. Box 3498	Palo Alto	CA	94303	650-328-4543
104	Anthony	Higgins	Play Ball!	East Shopping Center	422 Bay Road	Redwood City	CA	94026	650-368-1100
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive		Los Altos	CA	94022	650-776-3249
106	George	Watson	Watson & Son	1143 Carver Place		Mountain View	CA	94063	650-389-8789
107	Charles	Ream	Athletic Supplies	41 Jordan Avenue		Palo Alto	CA	94304	650-356-9876
108	Donald	Quinn	Quinn's Sports	587 Alvarado		Redwood City	CA	94063	650-544-8729
109	Jane	Miller	Sport Stuff	Mayfair Mart	7345 Ross Blvd.	Sunnyvale	CA	94086	408-723-8789
110	Roy	Jaeger	AA Athletics	520 Topaz Way		Redwood City	CA	94062	650-743-3611
111	Frances	Keyes	Sports Center	3199 Sterling Court		Sunnyvale	CA	94085	408-277-7245
112	Margaret	Lawson	Runners & Others	234 Wyandotte Way		Los Altos	CA	94022	650-887-7235
113	Lana	Beatty	Sportstown	654 Oak Grove		Menlo Park	CA	94025	650-356-9982
114	Frank	Albertson	Sporting Place	947 Waverly Place		Redwood City	CA	94062	650-886-6677
115	Alfred	Grant	Gold Medal Sports	776 Gary Avenue		Menlo Park	CA	94025	650-356-1123
116	Jean	Parmelee	Olympic City	1104 Spinosa Drive		Mountain View	CA	94040	650-534-8822
117	Arnold	Sipes	Kids Korner	850 Lytton Court		Redwood City	CA	94063	650-245-4578
118	Dick	Baxter	Blue Ribbon Sports	5427 College		Oakland	CA	94609	650-655-0011
119	Bob	Shorter	The Triathletes Club	2405 Kings Highway		Cherry Hill	NJ	08002	609-663-6079
120	Fred	Jewell	Century Pro Shop	6627 N. 17th Way		Phoenix	AZ	85016	602-265-8754
121	Jason	Wallack	City Sports	Lake Biltmore Mall	350 W. 23rd Street	Wilmington	DE	19898	302-366-7511
122	Cathy	O'Brian	The Sporting Life	543 Nassau Street		Princeton	NJ	08540	609-342-0054

customer_num	fname	lname	company	address1	address2	city	state	zip code	phone
123	Marvin	Hanlon	Bay Sports	10100 Bay Meadows Road	Suite 1020	Jacksonville	FL	32256	904-823-4239
124	Chris	Putnum	Putnum's Putters	4715 S.E. Adams Blvd	Suite 909C	Bartlesville	OK	74006	918-355-2074
125	James	Henry	Total Fitness Sports	1450 Common-wealth Avenue		Brighton	MA	02135	617-232-4159
126	Eileen	Neelie	Neelie's Discount Sports	2539 South Utica Street		Denver	CO	80219	303-936-7731
127	Kim	Satifer	Big Blue Bike Shop	Blue Island Square	12222 Gregory Street	Blue Island	NY	60406	312-944-5691
128	Frank	Lessor	Phoenix University	Athletic Department	1817 N. Thomas Road	Phoenix	AZ	85008	602-533-1817

items Table

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1001	1	HRO	1	250.00
1	1002	4	HSK	1	960.00
2	1002	3	HSK	1	240.00
1	1003	9	ANZ	1	20.00
2	1003	8	ANZ	1	840.00
3	1003	5	ANZ	5	99.00
1	1004	1	HRO	1	250.00
2	1004	2	HRO	1	126.00
3	1004	3	HSK	1	240.00
4	1004	1	HSK	1	800.00
1	1005	5	NRG	10	280.00
2	1005	5	ANZ	10	198.00
3	1005	6	SMT	1	36.00
4	1005	6	ANZ	1	48.00
1	1006	5	SMT	5	125.00
2	1006	5	NRG	5	140.00
3	1006	5	ANZ	5	99.00
4	1006	6	SMT	1	36.00
5	1006	6	ANZ	1	48.00
1	1007	1	HRO	1	250.00
2	1007	2	HRO	1	126.00
3	1007	3	HSK	1	240.00

item_num	order_num	stock_num	manu_code	quantity	total_price
4	1007	4	HRO	1	480.00
5	1007	7	HRO	1	600.00
1	1008	8	ANZ	1	840.00
2	1008	9	ANZ	5	100.00
1	1009	1	SMT	1	450.00
1	1010	6	SMT	1	36.00
2	1010	6	ANZ	1	48.00
1	1011	5	ANZ	5	99.00
1	1012	8	ANZ	1	840.00
2	1012	9	ANZ	10	200.00
1	1013	5	ANZ	1	19.80
2	1013	6	SMT	1	36.00
3	1013	6	ANZ	1	48.00
4	1013	9	ANZ	2	40.00
1	1014	4	HSK	1	960.00
2	1014	4	HRO	1	480.00
1	1015	1	SMT	1	450.00
1	1016	101	SHM	2	136.00
2	1016	109	PRC	3	90.00
3	1016	110	HSK	1	308.00
4	1016	114	PRC	1	120.00
1	1017	201	NKL	4	150.00
2	1017	202	KAR	1	230.00
3	1017	301	SHM	2	204.00
1	1018	307	PRC	2	500.00
2	1018	302	KAR	3	15.00
3	1018	110	PRC	1	236.00
4	1018	5	SMT	4	100.00
5	1018	304	HRO	1	280.00
1	1019	111	SHM	3	1499.97
1	1020	204	KAR	2	90.00
2	1020	301	KAR	4	348.00
1	1021	201	NKL	2	75.00
2	1021	201	ANZ	3	225.00
3	1021	202	KAR	3	690.00
4	1021	205	ANZ	2	624.00
1	1022	309	HRO	1	40.00
2	1022	303	PRC	2	96.00
3	1022	6	ANZ	2	96.00
1	1023	103	PRC	2	40.00
2	1023	104	PRC	2	116.00

item_num	order_num	stock_num	manu_code	quantity	total_price
3	1023	105	SHM	1	80.00
4	1023	110	SHM	1	228.00
5	1023	304	ANZ	1	170.00
6	1023	306	SHM	1	190.00

call_type Table

call_code	code_descr
B	billing error
D	damaged goods
I	incorrect merchandise sent
L	late shipment
O	other

orders Table

order_ num	order_ date	customer_ num	ship_ instruct	back log	po_num	ship_ date	ship_ weight	ship_ charge	paid_ date
1001	05/20/1998	104	express	n	B77836	06/01/ 1998	20.40	10.00	07/22/ 1998
1002	05/21/1998	101	PO on box; deliver back door only	n	9270	05/26/ 1998	50.60	15.30	06/03/ 1998
1003	05/22/1998	104	express	n	B77890	05/23/ 1998	35.60	10.80	06/14/ 1998
1004	05/22/1998	106	ring bell twice	y	8006	05/30/ 1998	95.80	19.20	
1005	05/24/1998	116	call before delivery	n	2865	06/09/ 1998	80.80	16.20	06/21/ 1998
1006	05/30/1998	112	after 10AM	y	Q13557		70.80	14.20	
1007	05/31/1998	117		n	278693	06/05/ 1998	125.90	25.20	
1008	06/07/1998	110	closed Monday	y	LZ230	07/06/ 1998	45.60	13.80	07/21/ 1998
1009	06/14/1998	111	door next to grocery	n	4745	06/21/ 1998	20.40	10.00	08/21/ 1998
1010	06/17/1998	115	deliver 776 King St. if no answer	n	429Q	06/29/ 1998	40.60	12.30	08/22/ 1998
1011	06/18/1998	104	express	n	B77897	07/03/ 1998	10.40	5.00	08/29/ 1998
1012	06/18/1998	117		n	278701	06/29/ 1998	70.80	14.20	
1013	06/22/1998	104	express	n	B77930	07/10/ 1998	60.80	12.20	07/31/ 1998

order_ num	order_ date	customer_ num	ship_ instruct	back log	po_num	ship_ date	ship_ weight	ship_ charge	paid_ date
1014	06/25/1998	106	ring bell, kick door loudly	n	8052	07/03/ 1998	40.60	12.30	07/10/ 1998
1015	06/27/1998	110	closed Mondays	n	MA003	07/16/ 1998	20.60	6.30	08/31/ 1998
1016	06/29/1998	119	delivery entrance off Camp St.	n	PC6782	07/12/ 1998	35.00	11.80	
1017	07/09/1998	120	North side of clubhouse	n	DM3543 31	07/13/ 1998	60.00	18.00	
1018	07/10/1998	121	SW corner of Biltmore Mall	n	S22942	07/13/ 1998	70.50	20.00	08/06/ 1998
1019	07/11/1998	122	closed til noon Mondays	n	Z55709	07/16/ 1998	90.00	23.00	08/06/ 1998
1020	07/11/1998	123	express	n	W2286	07/16/ 1998	14.00	8.50	09/20/ 1998
1021	07/23/1998	124	ask for Elaine	n	C3288	07/25/ 1998	40.00	12.00	08/22/ 1998
1022	07/24/1998	126	express	n	W9925	07/30/ 1998	15.00	13.00	09/02/ 1998
1023	07/24/1998	127	no deliveries after 3 p.m.	n	KF2961	07/30/ 1998	60.00	18.00	08/22/ 1998

stock Table

stock_num	manu_code	description	unit_rice	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
3	SHM	baseball bat	280.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
5	SMT	tennis racquet	25.00	each	each
5	ANZ	tennis racquet	19.80	each	each
6	SMT	tennis ball	36.00	case	24 cans/case
6	ANZ	tennis ball	48.00	case	24 cans/case
7	HRO	basketball	600.00	case	24/case
8	ANZ	volleyball	840.00	case	24/case
9	ANZ	volleyball net	20.00	each	each
101	PRC	bicycle tires	88.00	box	4/box

stock_num	manu_code	description	unit_rice	unit	unit_descr
101	SHM	bicycle tires	68.00	box	4/box
102	SHM	bicycle brakes	220.00	case	4 sets/case
102	PRC	bicycle brakes	480.00	case	4 sets/case
103	PRC	front derailleur	20.00	each	each
104	PRC	rear derailleur	58.00	each	each
105	PRC	bicycle wheels	53.00	pair	pair
105	SHM	bicycle wheels	80.00	pair	pair
106	PRC	bicycle stem	23.00	each	each
107	PRC	bicycle saddle	70.00	pair	pair
108	SHM	crankset	45.00	each	each
109	PRC	pedal binding	30.00	case	6 pairs/case
109	SHM	pedal binding	200.00	case	4 pairs/case
110	PRC	helmet	236.00	case	4/case
110	ANZ	helmet	244.00	case	4/case
110	SHM	helmet	228.00	case	4/case
110	HRO	helmet	260.00	case	4/case
110	HSK	helmet	308.00	case	4/case
111	SHM	10-spd, assmbld	499.99	each	each
112	SHM	12-spd, assmbld	549.00	each	each
113	SHM	18-spd, assmbld	685.90	each	each
114	PRC	bicycle gloves	120.00	case	10 pairs/case
201	NKL	golf shoes	37.50	each	each
201	ANZ	golf shoes	75.00	each	each
201	KAR	golf shoes	90.00	each	each
202	NKL	metal woods	174.00	case	2 sets/case
202	KAR	std woods	230.00	case	2 sets/case
203	NKL	irons/wedges	670.00	case	2 sets/case
204	KAR	putter	45.00	each	each
205	NKL	3 golf balls	312.00	case	24/case
205	ANZ	3 golf balls	312.00	case	24/case
205	HRO	3 golf balls	312.00	case	24/case
301	NKL	running shoes	97.00	each	each
301	HRO	running shoes	42.50	each	each
301	SHM	running shoes	102.00	each	each
301	PRC	running shoes	75.00	each	each
301	KAR	running shoes	87.00	each	each
301	ANZ	running shoes	95.00	each	each
302	HRO	ice pack	4.50	each	each
302	KAR	ice pack	5.00	each	each
303	PRC	socks	48.00	box	24 pairs/box
303	KAR	socks	36.00	box	24 pair/box

stock_num	manu_code	description	unit_rice	unit	unit_descr
304	ANZ	watch	170.00	box	10/box
304	HRO	watch	280.00	box	10/box
305	HRO	first-aid kit	48.00	case	4/case
306	PRC	tandem adapter	160.00	each	each
306	SHM	tandem adapter	190.00	each	each
307	PRC	infant jogger	250.00	each	each
308	PRC	twin jogger	280.00	each	each
309	HRO	ear drops	40.00	case	20/case
309	SHM	ear drops	40.00	case	20/case
310	SHM	kick board	80.00	case	10/case
310	ANZ	kick board	89.00	case	12/case
311	SHM	water gloves	48.00	box	4 pairs/box
312	SHM	racer goggles	96.00	box	12/box
312	HRO	racer goggles	72.00	box	12/box
313	SHM	swim cap	72.00	box	12/box
313	ANZ	swim cap	60.00	box	12/box

catalog Table

catalog _num	stock _num	manu _code	cat_descr	cat_picture	cat_advert
10001	1	HRO	Brown leather. Specify first baseman's or infield/outfield style. Specify right- or left-handed.	<BYTE value>	Your First Season's Baseball Glove
10002	1	HSK	Babe Ruth signature glove. Black leather. Infield/outfield style. Specify right- or left-handed.	<BYTE value>	All-Leather, Hand-Stitched, Deep-Pockets, Sturdy Webbing that Won't Let Go
10003	1	SMT	Catcher's mitt. Brown leather. Specify right- or left-handed.	<BYTE value>	A Sturdy Catcher's Mitt With the Perfect Pocket
10004	2	HRO	Jackie Robinson signature glove. Highest Professional quality, used by National League.	<BYTE value>	Highest Quality Ball Available, from the Hand-Stitching to the Robinson Signature
10005	3	HSK	Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.	<BYTE value>	High-Technology Design Expands the Sweet Spot
10006	3	SHM	Aluminum. Blue with black tape. 31", 20 oz or 22 oz; 32", 21 oz or 23 oz; 33", 22 oz or 24 oz.	<BYTE value>	Durable Aluminum for High School and Collegiate Athletes
10007	4	HSK	Norm Van Brocklin signature style.	<BYTE value>	Quality Pigskin with Norm Van Brocklin Signature
10008	4	HRO	NFL-Style pigskin.	<BYTE value>	Highest Quality Football for High School and Collegiate Competitions

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10009	5	NRG	Graphite frame. Synthetic strings.	<BYTE value>	Wide Body Amplifies Your Natural Abilities by Providing More Power Through Aerodynamic Design
10010	5	SMT	Aluminum frame. Synthetic strings.	<BYTE value>	Mid-Sized Racquet for the Improving Player
10011	5	ANZ	Wood frame, cat-gut strings.	<BYTE value>	Antique Replica of Classic Wooden Racquet Built with Cat-Gut Strings
10012	6	SMT	Soft yellow color for easy visibility in sunlight or artificial light.	<BYTE value>	High-Visibility Tennis, Day or Night
10013	6	ANZ	Pro-core. Available in neon yellow, green, and pink.	<BYTE value>	Durable Construction Coupled with the Brightest Colors Available
10014	7	HRO	Indoor. Classic NBA style. Brown leather.	<BYTE value>	Long-Life Basketballs for Indoor Gymnasiums
10015	8	ANZ	Indoor. Finest leather. Professional quality.	<BYTE value>	Professional Volleyballs for Indoor Competitions
10016	9	ANZ	Steel eyelets. Nylon cording. Double-stitched. Sanctioned by the National Athletic Congress.	<BYTE value>	Sanctioned Volleyball Netting for Indoor Professional and Collegiate Competition
10017	101	PRC	Reinforced, hand-finished tubular. Polyurethane belted. Effective against punctures. Mixed tread for super wear and road grip.	<BYTE value>	Ultimate in Puncture Protection, Tires Designed for In-City Riding
10018	101	SHM	Durable nylon casing with butyl tube for superior air retention. Center-ribbed tread with herringbone side. Coated sidewalls resist abrasion.	<BYTE value>	The Perfect Tire for Club Rides or Training
10019	102	SHM	Thrust bearing and coated pivot washer/ spring sleeve for smooth action. Slotted levers with soft gum hoods. Two-tone paint treatment. Set includes calipers, levers, and cables.	<BYTE value>	Thrust-Bearing and Spring-Sleeve Brake Set Guarantees Smooth Action
10020	102	PRC	Computer-aided design with low-profile pads. Cold-forged alloy calipers and beefy caliper bushing. Aero levers. Set includes calipers, levers, and cables.	<BYTE value>	Computer Design Delivers Rigid Yet Vibration-Free Brakes
10021	103	PRC	Compact leading-action design enhances shifting. Deep cage for super-small granny gears. Extra strong construction to resist off-road abuse.	<BYTE value>	Climb Any Mountain: ProCycle's Front Derailleur Adds Finesse to Your ATB
10022	104	PRC	Floating trapezoid geometry with extra thick parallelogram arms. 100-tooth capacity. Optimum alignment with any freewheel.	<BYTE value>	Computer-Aided Design Engineers 100-Tooth Capacity Into ProCycle's Rear Derailleur

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10023	105	PRC	Front wheels laced with 15g spokes in a 3-cross pattern. Rear wheels laced with 14g spikes in a 3-cross pattern.	<BYTE value>	Durable Training Wheels That Hold True Under Toughest Conditions
10024	105	SHM	Polished alloy. Sealed-bearing, quick-release hubs. Double-butted. Front wheels are laced 15g/2-cross. Rear wheels are laced 15g/3-cross.	<BYTE value>	Extra Lightweight Wheels for Training or High-Performance Touring
10025	106	PRC	Hard anodized alloy with pearl finish. 6mm hex bolt hardware. Available in lengths of 90-140mm in 10mm increments.	<BYTE value>	ProCycle Stem with Pearl Finish
10026	107	PRC	Available in three styles: Men's racing; Men's touring; and Women's. Anatomical gel construction with lycra cover. Black or black/hot pink.	<BYTE value>	The Ultimate In Riding Comfort, Lightweight With Anatomical Support
10027	108	SHM	Double or triple crankset with choice of chainrings. For double crankset, chainrings from 38-54 teeth. For triple crankset, chainrings from 24-48 teeth.	<BYTE value>	Customize Your Mountain Bike With Extra-Durable Crankset
10028	109	PRC	Steel toe clips with nylon strap. Extra wide at buckle to reduce pressure.	<BYTE value>	Classic Toeclip Improved to Prevent Soreness at Clip Buckle
10029	109	SHM	Ingenious new design combines button on sole of shoe with slot on a pedal plate to give riders new options in riding efficiency. Select full or partial locking. Four plates mean both top and bottom of pedals are slotted—no fishing around when you want to engage full power. Fast unlocking ensures safety when maneuverability is paramount.	<BYTE value>	Ingenious Pedal/Clip Design Delivers Maximum Power and Fast Unlocking
10030	110	PRC	Super-lightweight. Meets both ANSI and Snell standards for impact protection. 7.5 oz. Quick-release shadow buckle.	<BYTE value>	Feather-Light, Quick-Release, Maximum Protection Helmet
10031	110	ANZ	No buckle so no plastic touches your chin. Meets both ANSI and Snell standards for impact protection. 7.5 oz. Lycra cover.	<BYTE value>	Minimum Chin Contact, Feather-Light, Maximum Protection Helmet
10032	110	SHM	Dense outer layer combines with softer inner layer to eliminate the mesh cover, no snagging on brush. Meets both ANSI and Snell standards for impact protection. 8.0 oz.	<BYTE value>	Mountain Bike Helmet: Smooth Cover Eliminates the Worry of Brush Snags But Delivers Maximum Protection

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10033	110	HRO	Newest ultralight helmet uses plastic shell. Largest ventilation channels of any helmet on the market. 8.5 oz.	<BYTE value>	Lightweight Plastic with Vents Assures Cool Comfort Without Sacrificing Protection
10034	110	HSK	Aerodynamic (teardrop) helmet covered with anti-drag fabric. Credited with shaving 2 seconds/mile from winner's time in Tour de France time-trial. 7.5 oz.	<BYTE value>	Teardrop Design Used by Yellow Jerseys, You Can Time the Difference
10035	111	SHM	Light-action shifting 10 speed. Designed for the city commuter with shock-absorbing front fork and drilled eyelets for carry-all racks or bicycle trailers. Internal wiring for generator lights. 33 lbs.	<BYTE value>	Fully Equipped Bicycle Designed for the Serious Commuter Who Mixes Business With Pleasure
10036	112	SHM	Created for the beginner enthusiast. Ideal for club rides and light touring. Sophisticated triple-butted frame construction. Precise index shifting. 28 lbs.	<BYTE value>	We Selected the Ideal Combination of Touring Bike Equipment, then Turned It Into This Package Deal: High-Performance on the Roads, Maximum Pleasure Everywhere
10037	113	SHM	Ultra-lightweight. Racing frame geometry built for aerodynamic handlebars. Cantilever brakes. Index shifting. High-performance gearing. Quick-release hubs. Disk wheels. Bladed spokes.	<BYTE value>	Designed for the Serious Competitor, The Complete Racing Machine
10038	114	PRC	Padded leather palm and stretch mesh merged with terry back; Available in tan, black, and cream. Sizes S, M, L, XL.	<BYTE value>	Riding Gloves for Comfort and Protection
10039	201	NKL	Designed for comfort and stability. Available in white & blue or white & brown. Specify size.	<BYTE value>	Full-Comfort, Long-Wearing Golf Shoes for Men and Women
10040	201	ANZ	Guaranteed waterproof. Full leather upper. Available in white, bone, brown, green, and blue. Specify size.	<BYTE value>	Waterproof Protection Ensures Maximum Comfort and Durability In All Climates
10041	201	KAR	Leather and leather mesh for maximum ventilation. Waterproof lining to keep feet dry. Available in white and gray or white and ivory. Specify size.	<BYTE value>	Karsten's Top Quality Shoe Combines Leather and Leather Mesh
10042	202	NKL	Complete starter set utilizes gold shafts. Balanced for power.	<BYTE value>	Starter Set of Woods, Ideal for High School and Collegiate Classes
10043	202	KAR	Full set of woods designed for precision control and power performance.	<BYTE value>	High-Quality Woods Appropriate for High School Competitions or Serious Amateurs

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10044	203	NKL	Set of eight irons includes 3 through 9 irons and pitching wedge. Originally priced at $489.00.	<BYTE value>	Set of Irons Available From Factory at Tremendous Savings: Discontinued Line
10045	204	KAR	Ideally balanced for optimum control. Nylon-covered shaft.	<BYTE value>	High-Quality Beginning Set of Irons Appropriate for High School Competitions
10046	205	NKL	Fluorescent yellow.	<BYTE value>	Long Drive Golf Balls: Fluorescent Yellow
10047	205	ANZ	White only.	<BYTE value>	Long Drive Golf Balls: White
10048	205	HRO	Combination fluorescent yellow and standard white.	<BYTE value>	HiFlier Golf Balls: Case Includes Fluorescent Yellow and Standard White
10049	301	NKL	Super shock-absorbing gel pads disperse vertical energy into a horizontal plane for extraordinary cushioned comfort. Great motion control. Men's only. Specify size.	<BYTE value>	Maximum Protection For High-Mileage Runners
10050	301	HRO	Engineered for serious training with exceptional stability. Fabulous shock absorption. Great durability. Specify men's/women's, size.	<BYTE value>	Pronators and Supinators Take Heart: A Serious Training Shoe For Runners Who Need Motion Control
10051	301	SHM	For runners who log heavy miles and need a durable, supportive, stable platform. Mesh/synthetic upper gives excellent moisture dissipation. Stability system uses rear antipronation platform and forefoot control plate for extended protection during high-intensity training. Specify men's/women's size.	<BYTE value>	The Training Shoe Engineered for Marathoners and Ultra-Distance Runners
10052	301	PRC	Supportive, stable racing flat. Plenty of forefoot cushioning with added motion control. Women's only. D widths available. Specify size.	<BYTE value>	A Woman's Racing Flat That Combines Extra Forefoot Protection With a Slender Heel
10053	301	KAR	Anatomical last holds your foot firmly in place. Feather-weight cushioning delivers the responsiveness of a racing flat. Specify men's/women's size.	<BYTE value>	Durable Training Flat That Can Carry You Through Marathon Miles
10054	301	ANZ	Cantilever sole provides shock absorption and energy rebound. Positive traction shoe with ample toe box. Ideal for runners who need a wide shoe. Available in men's and women's. Specify size.	<BYTE value>	Motion Control, Protection, and Extra Toebox Room
10055	302	KAR	Reusable ice pack with velcro strap. For general use. Velcro strap allows easy application to arms or legs.	<BYTE value>	Finally, an Ice Pack for Achilles Injuries and Shin Splints That You Can Take to the Office

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10056	303	PRC	Neon nylon. Perfect for running or aerobics. Indicate color: Fluorescent pink, yellow, green, and orange.	<BYTE value>	Knock Their Socks Off With YOUR Socks
10057	303	KAR	100% nylon blend for optimal wicking and comfort. We've taken out the cotton to eliminate the risk of blisters and reduce the opportunity for infection. Specify men's or women's.	<BYTE value>	100% Nylon Blend Socks - No Cotton
10058	304	ANZ	Provides time, date, dual display of lap/cumulative splits, 4-lap memory, 10 hr count-down timer, event timer, alarm, hour chime, waterproof to 50m, velcro band.	<BYTE value>	Athletic Watch w/4-Lap Memory
10059	304	HRO	Split timer, waterproof to 50m. Indicate color: Hot pink, mint, green, space black.	<BYTE value>	Waterproof Triathlete Watch In Competition Colors
10060	305	HRO	Contains ace bandage, anti-bacterial cream, alcohol cleansing pads, adhesive bandages of assorted sizes, and instant-cold pack.	<BYTE value>	Comprehensive First-Aid Kit Essential for Team Practices, Team Traveling
10061	306	PRC	Converts a standard tandem bike into an adult/child bike. User-tested assembly instructions	<BYTE value>	Enjoy Bicycling With Your Child on a Tandem; Make Your Family Outing Safer
10062	306	SHM	Converts a standard tandem bike into an adult/child bike. Lightweight model.	<BYTE value>	Consider a Touring Vacation for the Entire Family: A Lightweight, Touring Tandem for Parent and Child
10063	307	PRC	Allows mom or dad to take the baby out too. Fits children up to 21 pounds. Navy blue with black trim.	<BYTE value>	Infant Jogger Keeps A Running Family Together
10064	308	PRC	Allows mom or dad to take both children! Rated for children up to 18 pounds.	<BYTE value>	As Your Family Grows, Infant Jogger Grows With You
10065	309	HRO	Prevents swimmer's ear.	<BYTE value>	Swimmers Can Prevent Ear Infection All Season Long
10066	309	SHM	Extra-gentle formula. Can be used every day for prevention or treatment of swimmer's ear.	<BYTE value>	Swimmer's Ear Drops Specially Formulated for Children
10067	310	SHM	Blue heavy-duty foam board with Shimara or team logo.	<BYTE value>	Exceptionally Durable, Compact Kickboard for Team Practice
10068	310	ANZ	White. Standard size.	<BYTE value>	High-Quality Kickboard
10069	311	SHM	Swim gloves. Webbing between fingers promotes strengthening of arms. Cannot be used in competition.	<BYTE value>	Hot Training Tool - Webbed Swim Gloves Build Arm Strength and Endurance

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10070	312	SHM	Hydrodynamic egg-shaped lens. Ground-in anti-fog elements; Available in blue or smoke.	<BYTE value>	Anti-Fog Swimmer's Goggles: Quantity Discount
10071	312	HRO	Durable competition-style goggles. Available in blue, grey, or white.	<BYTE value>	Swim Goggles: Traditional Rounded Lens For Greater Comfort
10072	313	SHM	Silicone swim cap. One size. Available in white, silver, or navy. Team Logo Imprinting Available.	<BYTE value>	Team Logo Silicone Swim Cap
10073	314	ANZ	Silicone swim cap. Squared-off top. One size. White	<BYTE value>	Durable Squared-off Silicone Swim Cap
10074	315	HRO	Re-usable ice pack. Store in the freezer for instant first-aid. Extra capacity to accommodate water and ice.	<BYTE value>	Water Compartment Combines With Ice to Provide Optimal Orthopedic Treatment

cust_calls Table

customer_num	call_dtime	user_id	call_code	call_descr	res_dtime	res_descr
106	1998-06-12 8:20	maryj	D	Order was received, but two of the cans of ANZ tennis balls within the case were empty.	1998-06-12 8:25	Authorized credit for two cans to customer, issued apology. Called ANZ buyer to report the QA problem.
110	1998-07-07 10:24	richc	L	Order placed one month ago (6/7) not received.	1998-07-07 10:30	Checked with shipping (Ed Smith). Order sent yesterday-we were waiting for goods from ANZ. Next time will call with delay if necessary.
119	1998-07-01 15:00	richc	B	Bill does not reflect credit from previous order.	1998-07-02 8:21	Spoke with Jane Akant in Finance. She found the error and is sending new bill to customer.
121	1998-07-10 14:05	maryj	O	Customer likes our merchandise. Requests that we stock more types of infant joggers. Will call back to place order.	1998-07-10 14:06	Sent note to marketing group of interest in infant joggers.
127	1998-07-31 14:30	maryj	I	Received Hero watches (item # 304) instead of ANZ watches.		Sent memo to shipping to send ANZ item 304 to customer and pickup HRO watches. Should be done tomorrow, 8/1.
116	1997-11-28 13:34	mannyn	I	Received plain white swim caps (313 ANZ) instead of navy with team logo (313 SHM).	1997-11-28 16:47	Shipping found correct case in warehouse and express mailed it in time for swim meet.

customer_num	call_dtime	user_id	call_code	call_descr	res_dtime	res_descr
116	1997-12-21 11:24	mannyn	I	Second complaint from this customer! Received two cases right-handed outfielder gloves (1 HRO) instead of one case lefties.	1997-12-27 08:19	Memo to shipping (Ava Brown) to send case of left-handed gloves, pick up wrong case; memo to billing requesting 5% discount to placate customer due to second offense and lateness of resolution because of holiday.

manufact Table

manu_code	manu_name	lead_time
ANZ	Anza	5
HSK	Husky	5
HRO	Hero	4
NRG	Norge	7
SMT	Smith	3
SHM	Shimara	30
KAR	Karsten	21
NKL	Nikolus	8
PRC	ProCycle	9

state Table

code	sname	code	sname
AK	Alaska	MT	Montana
AL	Alabama	NE	Nebraska
AR	Arkansas	NC	North Carolina
AZ	Arizona	ND	North Dakota
CA	California	NH	New Hampshire
CT	Connecticut	NJ	New Jersey
CO	Colorado	NM	New Mexico
DC	Washington, D.C.	NV	Nevada
DE	Delaware	NY	New York
FL	Florida	OH	Ohio
GA	Georgia	OK	Oklahoma
HI	Hawaii	OR	Oregon
IA	Iowa	PA	Pennsylvania
ID	Idaho	PR	Puerto Rico
IL	Illinois	RI	Rhode Island
IN	Indiana	SC	South Carolina
KY	Kentucky	TN	Tennessee
LA	Louisiana	TX	Texas

code	sname	code	sname
MA	Massachusetts	UT	Utah
MD	Maryland	VA	Virginia
ME	Maine	VT	Vermont
MI	Michigan	WA	Washington
MN	Minnesota	WI	Wisconsin
MO	Missouri	WV	West Virginia
MS	Mississippi	WY	Wyoming

Appendix B. The sales_demo and superstores_demo Databases

In addition to the **stores_demo** database that is described in detail in Appendix A, "The stores_demo Database," on page A-1, IBM Informix products include the following demonstration databases:

- **Extended Parallel Server**: The **sales_demo** database illustrates a dimensional schema for data-warehousing applications.
- **IBM Informix**: The **superstores_demo** database illustrates an object-relational schema.

This appendix contains information about the structures of these two demonstration databases.

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*.

The sales_demo Database (XPS)

Your database server product contains SQL scripts for the **sales_demo** dimensional database. The **sales_demo** database provides an example of a simple data-warehousing environment and works in conjunction with the **stores_demo** database. The scripts for the **sales_demo** database create new tables and add extra rows to the **items** and **orders** tables of **stores_demo**.

To create the **sales_demo** database, you must first create the **stores_demo** database with the logging option. After you create the **stores_demo** database, you can execute the scripts that create and load the **sales_demo** database from DB–Access. The files are named **createdw.sql** and **loaddw.sql**.

Dimensional Model of the sales_demo Database

Figure B-1 on page B-2 gives an overview of the tables in the **sales_demo** database.

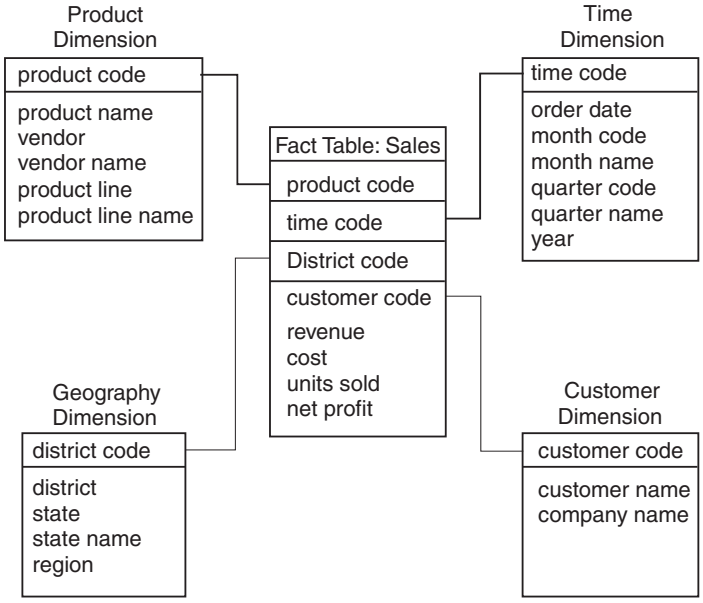


Figure B-1. The sales_demo Dimensional Data Model

For information about how to create and populate the **sales_demo** database, see the *IBM Informix DB-Access User's Guide*. For information about how to design and implement dimensional databases, see the *IBM Informix Database Design and Implementation Guide*. For information about the **stores_demo** database, see Appendix A, "The stores_demo Database," on page A-1

The next section describes the schema of these five tables, and identifies the column in the fact table that logically associates each dimension table to the fact table.

Structure of the sales_demo Tables

The **sales_demo** database includes the following tables:

- **customer**
- **geography**
- **product**
- **sales**
- **time**

The tables are listed alphabetically, not in the order in which they are created. The **customer**, **geography**, **product**, and **time** tables are the dimensions for the **sales** fact table.

The **sales_demo** database is not an ANSI-compliant database.

The following sections describe the column names, data types, and column descriptions for each table. A SERIAL field serves as the primary key for the **district_code** column of the **geography** table. The primary and foreign key relationships that exist between the fact (**sales**) table and its dimension tables are not defined, however, because data-loading performance improves dramatically when the database server does not enforce constraint checking.

The customer Table

The **customer** table contains information about sales customers. Table B-1 shows the columns of the **customer** table.

Table B-1. The customer Table

Name	Type	Description
customer_code	INTEGER	Customer code
customer_name	CHAR(31)	Customer name
company_name	CHAR(20)	Company name

The geography Table

The **geography** table contains information about the sales district and region. Table B-2 shows the columns of the **geography** table.

Table B-2. The geography Table

Name	Type	Description
district_code	SERIAL	District code
district_name	CHAR(15)	District name
state_code	CHAR(2)	State code
state_name	CHAR(18)	State name
region	SMALLINT	Region name

The product Table

The **product** table contains information about the products sold through the data warehouse. Table B-3 shows the columns of the **product** table.

Table B-3. The product Table

Name	Type	Description
product_code	INTEGER	Product code
product_name	CHAR(31)	Product name
vendor_code	CHAR(3)	Vendor code
vendor_name	CHAR(15)	Vendor name
product_line_code	SMALLINT	Product line code
product_line_name	CHAR(15)	Name of product line

The sales Table

The **sales** fact table contains information about product sales and has a pointer to each dimension table. For example, the **customer_code** column references the **customer** table, the **district_code** column references the **geography** table, and so on. The **sales** table also contains the measures for the units sold, revenue, cost, and net profit. Table B-4 shows the columns of the **sales** table.

Table B-4. The sales Table

Name	Type	Description
customer_code	INTEGER	Customer code (references **customer**)

Table B-4. The sales Table (continued)

Name	Type	Description
district_code	SMALLINT	District code (references **geography**)
time_code	INTEGER	Time code (references **time**)
product_code	INTEGER	Product code (references **product**)
units_sold	SMALLINT	Number of units sold
revenue	MONEY(8,2)	Amount of sales revenue
cost	MONEY(8,2)	Cost of sale
net_profit	MONEY(8,2)	Net profit of sale

The time Table

The **time** table contains time information about the sale. Table B-5 shows the columns of the **time** table.

Table B-5. The time Table

Name	Type	Description
time_code	INTEGER	Time code
order_date	DATE	Order date
month_code	SMALLINT	Month code
month_name	CHAR(10)	Name of month
quarter_code	SMALLINT	Quarter code
quarter_name	CHAR(10)	Name of quarter
year	INTEGER	Year

The superstores_demo Database

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.

This section provides the following **superstores_demo** information:

- The structure of all the tables in the **superstores_demo** database
- A list and definition of the extended data types that **superstores_demo** uses
- A map of table hierarchies
- The primary-foreign key relationships among the columns in the database tables

For information about how to create and populate the **superstores_demo** database, see the *IBM Informix DB-Access User's Guide*. For information about how to work with object-relational databases, see the *IBM Informix Database Design and Implementation Guide*. For information about the **stores_demo** database on which **superstores_demo** is based, see Appendix A, "The stores_demo Database," on page A-1.

Structure of the superstores_demo Tables

The **superstores_demo** database includes the following tables. Although many tables have the same name as **stores_demo** tables, they are different. The tables are listed alphabetically, not in the order in which they are created.

- **call_type**
- **catalog**
- **cust_calls**
- **customer**
  - **retail_customer** (*new*)
  - **whlsale_customer** (*new*)
- **items**
- **location** (*new*)
  - **location_non_us** (*new*)
  - **location_us** (*new*)
- **manufact**
- **orders**
- **region** (*new*)
- **sales_rep** (*new*)
- **state**
- **stock**
- **stock_discount** (*new*)
- **units** (*new*)

This section lists the names, data types, and descriptions of the columns for each table in the **superstores_demo** database. The unique identifying value for each table (primary key) is shaded. Columns that represent extended data types are explained in "User-Defined Routines and Extended Data Types" on page B-12. Primary-foreign key relationships between the tables are outlined in "Referential Relationships" on page B-14.

The call_type Table

The call codes associated with customer calls are stored in the **call_type** table. Table B-6 shows the columns of the **call_type** table.

Table B-6. The call_type Table

Name	Type	Description
call_code	CHAR(1)	Call code
codel_descr	CHAR (30)	Description of call code

The catalog Table

The **catalog** table describes each item in stock. Retail stores use this table when placing orders with the distributor. Table B-7 shows the columns of the **catalog** table.

Table B-7. The catalog Table

Name	Type	Description
catalog_num	SERIAL(1001)	System-generated catalog number

Table B-7. The catalog Table (continued)

Name	Type	Description
stock_num	SMALLINT	Distributor stock number (foreign key to **stock** table)
manu_code	CHAR(3)	Manufacturer code (foreign key to **stock** table)
unit	CHAR(4)	Unit by which item is ordered (foreign key to **stock** table)
advert	ROW (picture BLOB, caption LVARCHAR)	Picture of item and caption
advert_descr	CLOB	Tag line underneath picture

The cust_calls Table

All customer calls for information about orders, shipments, or complaints are logged. The **cust_calls** table contains information about these types of customer calls. Table B-8 shows the columns of the **cust_calls** table.

Table B-8. The cust_calls Table

Name	Type	Description
customer_num	INTEGER	Customer number (foreign key to **customer** table)
call_dtime	DATETIME YEAR TO MINUTE	Date and time call received
user_id	CHAR(18)	Name of person logging call (default is user login name)
call_code	CHAR(1)	Type of call (foreign key to **call_type** table)
call_descr	CHAR(240)	Description of call
res_dtime	DATETIME YEAR TO MINUTE	Date and time call resolved
res_descr	CHAR(240)	Description of how call was resolved

The customer, retail_customer, and whlsale_customer Tables

In this hierarchy, **retail_customer** and **whlsale_customer** are subtables that are created under the **customer** supertable, as Figure B-2 on page B-14 shows.

For information about table hierarchies, see the *IBM Informix Database Design and Implementation Guide*.

The customer Table:

The **customer** table contains information about the retail stores that place orders from the distributor. Table B-9 shows the columns of the **customer** table.

Table B-9. The customer Table

Name	Type	Description
customer_num	SERIAL	Unique customer identifier
customer_type	CHAR(1)	Code to indicate type of customer: `R` = retail `W` = wholesale
customer_name	name_t	Name of customer

Table B-9. The customer Table (continued)

Name	Type	Description
customer_loc	INTEGER	Location of customer (foreign key to **location** table)
contact_dates	LIST(DATETIME YEAR TO DAY NOT NULL)	Dates of contact with customer
cust_discount	percent	Customer discount
credit_status	CHAR(1)	Customer credit status: `D` = deadbeat `L` = lost `N` = new `P` = preferred `R` = regular

The retail_customer Table:

The **retail_customer** table contains general information about retail customers. Table B-10 shows the columns of the **retail_customer** table.

Table B-10. The retail_customer Table

Name	Type	Description
customer_num	SERIAL	Unique customer identifier
customer_type	CHAR(1)	Code to indicate type of customer: `R` = retail `W` = wholesale
customer_name	name_t	Name of customer
customer_loc	INTEGER	Location of customer
contact_dates	LIST(DATETIME YEAR TO DAY NOT NULL)	Dates of contact with customer
cust_discount	percent	Customer discount
credit_status	CHAR(1)	Customer credit status: `D` = deadbeat `L` = lost `N` = new `P` = preferred `R` = regular
credit_num	CHAR(19)	Credit card number
expiration	DATE	Expiration data of credit card

The whlsale_customer Table:

The **whlsale_customer** table contains general information about wholesale customers. Table B-11 shows the columns of the **whlsale_customer** table.

Table B-11. The whlsale_customer Table

Name	Type	Description
customer_num	SERIAL	Unique customer identifier
customer_type	CHAR(1)	Code to indicate type of customer: `R` = retail `W` = wholesale
customer_name	name_t	Name of customer
customer_loc	INTEGER	Location of customer
contact_dates	LIST(DATETIME YEAR TO DAY NOT NULL)	Dates of contact with customer
cust_discount	percent	Customer discount
credit_status	CHAR(1)	Customer credit status: `D` = deadbeat `L` = lost `N` = new `P` = preferred `R` = regular

Table B-11. The whlsale_customer Table (continued)

Name	Type	Description
resale_license	CHAR(15)	Resale license number
terms_net	SMALLINT	Net term in days

The items Table

An order can include one or more items. One row exists in the **items** table for each item in an order. Table B-12 shows the columns of the **items** table.

Table B-12. The items Table

Name	Type	Description
item_num	SMALLINT	Sequentially assigned item number for an order
order_num	INT8	Order number (foreign key to **orders** table)
stock_num	SMALLINT	Stock number for item (foreign key to **stock** table)
manu_code	CHAR(3)	Manufacturer code for item ordered (foreign key to **stock** table)
unit	CHAR(4)	Unit by which item is ordered (foreign key to **stock** table)
quantity	SMALLINT	Quantity ordered (value must be > 1)
item_subtotal	MONEY(8,2)	Quantity ordered * unit price = total price of item

The location, location_non_us, and location_us Tables

In this hierarchy, **location_non_us** and **location_us** are subtables that are created under the **location** supertable, as shown in the diagram in "Table Hierarchies" on page B-14. For information about table hierarchies, see the *IBM Informix Database Design and Implementation Guide*.

The location Table

The **location** table contains general information about the locations (addresses) that the database tracks. Table B-13 shows the columns of the **location** table.

Table B-13. The location Table

Name	Type	Description
location_id	SERIAL	Unique identifier for location
loc_type	CHAR(2)	Code to indicate type of location
company	VARCHAR(20)	Name of company
street_addr	LIST(VARCHAR(25) NOT NULL)	Street address
city	VARCHAR(25)	City for address
country	VARCHAR(25)	Country for address

The location_non_us Table

The **location_non_us** table contains specific address information for locations (addresses) that are outside the United States. Table B-14 shows the columns of the **location_non_us** table.

Table B-14. The location_non_us Table

Name	Type	Description
location_id	SERIAL	Unique identifier for location
loc_type	CHAR(2)	Code to indicate type of location
company	VARCHAR(20)	Name of company
street_addr	LIST(VARCHAR(25) NOT NULL)	Street address
city	VARCHAR(25)	City for address
country	VARCHAR(25)	Country for address
province_code	CHAR(2)	Province code
zipcode	CHAR(9)	Zip code
phone	CHAR(15)	Phone number

The location_us Table

The **location_us** table contains specific address information for locations (addresses) that are in the United States. Table B-15 shows the columns of the **location_us** table.

Table B-15. The location_us Table

Name	Type	Description
location_id	SERIAL	Unique identifier for location
loc_type	CHAR(2)	Code to indicate type of location
company	VARCHAR(20)	Name of company
street_addr	LIST(VARCHAR(25) NOT NULL)	Street address
city	VARCHAR(25)	City for address
country	VARCHAR(25)	Country for address
state_code	CHAR(2)	State code (foreign key to **state** table)
zip	CHAR(9)	Zip code
phone	CHAR(15)	Phone number

The manufact Table

Information about the manufacturers whose sporting goods are handled by the distributor is stored in the **manufact** table. Table B-16 shows the columns of the **manufact** table.

Table B-16. The manufact Table

Name	Type	Description
manu_code	CHAR(3)	Manufacturer code
manu_name	VARCHAR(15)	Name of manufacturer

Table B-16. The manufact Table (continued)

Name	Type	Description
lead_time	INTERVAL DAY(3) TO DAY	Lead time for shipment of orders
manu_loc	INTEGER	Manufacturer location (foreign key to **location** table)
manu_account	CHAR(32)	Distributor account number with manufacturer
account_status	CHAR(1)	Status of account with manufacturer
terms_net	SMALLINT	Distributor terms with manufacturer (in days)
discount	percent	Distributor volume discount with manufacturer

The orders Table

The **orders** table contains information about orders placed by the customers of the distributor. Table B-17 shows the columns of the **orders** table.

Table B-17. The orders Table

Name	Type	Description
order_num	SERIAL8(1001)	System-generated order number
order_date	DATE	Date order entered
customer_num	INTEGER	Customer number (foreign key to **customer** table)
shipping	ship_t	Special shipping instructions
backlog	BOOLEAN	Indicates order cannot be filled because the item is back ordered
po_num	CHAR(10)	Customer purchase order number
paid_date	DATE	Date order paid

The region Table

The **region** table contains information about the sales regions for the distributor. Table B-18 shows the columns of the **region** table.

Table B-18. The region Table

Name	Type	Description
region_num	SERIAL	System-generated region number
region_name	VARCHAR(20) UNIQUE	Name of sales region
region_loc	INTEGER	Location of region office (foreign key to **location** table)

The sales_rep Table

The **sales_rep** table contains information about the sales representatives for the distributor. Table B-19 on page B-11 shows the columns of the **sales_rep** table.

Table B-19. The sales_rep Table

Name	Type	Description
rep_num	SERIAL(101)	System-generated sales rep number
name	name_t	Name of sales rep
region_num	INTEGER	Region in which sales rep works (foreign key to the **region** table)
home_office	BOOLEAN	Home office location of sales rep
sales	SET(ROW (month DATETIME YEAR TO MONTH, amount MONEY) NOT NULL)	Amount of monthly sales for rep
commission	percent	Commission rate for sales rep

The state Table

The **state** table contains the names and postal abbreviations, and sales tax information, for the 50 states of the United States. Table B-20 shows the columns of the **state** table.

Table B-20. The state Table

Name	Type	Description
code	CHAR(2)	State code
sname	CHAR(15)	State name
sales_tax	percent	State sales tax

The stock Table

The **stock** table is a catalog of the items sold by the distributor. Table B-21 shows the columns of the **stock** table.

Table B-21. The stock Table

Name	Type	Description
stock_num	SMALLINT	Stock number that identifies type of item
manu_code	CHAR(3)	Manufacturer code (foreign key to **manufact**)
unit	CHAR(4)	Unit by which item is ordered
description	VARCHAR(15)	Description of item
unit_price	MONEY(6,2)	Unit price
min_reord_qty	SMALLINT	Minimum reorder quantity
min_inv_qty	SMALLINT	Quantity of stock below which item should be reordered
manu_item_num	CHAR(20)	Manufacturer item number
unit_cost	MONEY(6,2)	Distributor cost per unit of item from manufacturer
status	CHAR(1)	Status of item: `A` = active `D` = discontinued `N` = no order
bin_num	INTEGER	Bin number

Table B-21. The stock Table (continued)

Name	Type	Description
qty_on_hand	SMALLINT	Quantity in stock
bigger_unit	CHAR(4)	Stock unit for next larger unit (for same stock_num and manu_code)
per_bigger_unit	SMALLINT	How many of this item in bigger_unit

The stock_discount Table

The **stock_discount** table contains information about stock discounts. (There is no primary key). Table B-22 shows the columns of the **stock_discount** table.

Table B-22. The stock_discount Table

Name	Type	Description
discount_id	SERIAL	System-generated discount identifier
stock_num	SMALLINT	Distributor stock number (part of foreign key to **stock** table)
manu_code	CHAR(3)	Manufacturer code (part of foreign key to **stock** table)
unit	CHAR(4)	Unit by which item is ordered (each, pair, case, and so on) (foreign key to **units** table; part of foreign key to **stock** table)
unit_discount	percent	Unit discount during sale period
start_date	DATE	Discount start date
end_date	DATE	Discount end date

The units Table

The **units** table contains information about the units in which the inventory items can be ordered. Each item in the **stock** table is available in one or more types of container. Table B-23 shows the columns of the **units** table.

Table B-23. The units Table

Name	Type	Description
unit_name	CHAR(4)	Units by which an item is ordered (each, pair, case, box)
unit_descr	VARCHAR(15)	Description of units

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

This section lists the extended data types and UDRs created for the **superstores_demo** database. 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

Figure B-2 shows how the hierarchical tables of the **superstores_demo** database are related. See "The customer and location Tables" on page B-16 for an explanation of the foreign key and primary relationships between those two tables, which are indicated by shaded arrows that point from the **customer.custnum** and **customer.loc** columns to the **location.location_id** columns in the following diagram.

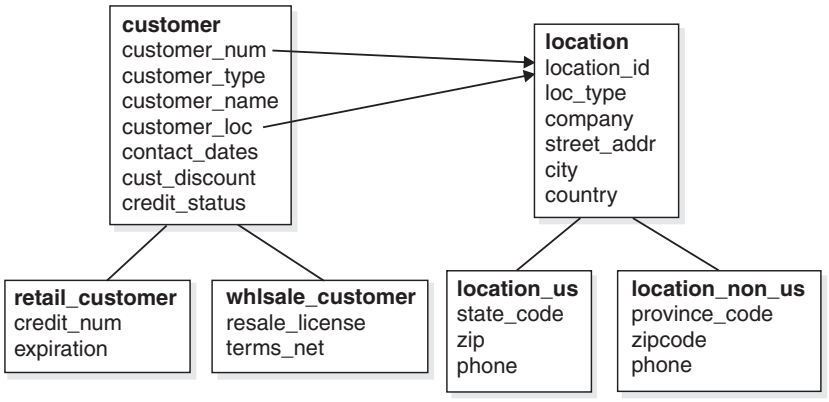


Figure B-2. Hierarchies of superstores_demo Tables

Referential Relationships

The tables of the **superstores_demo** database are linked by the primary-foreign key relationships that are identified in this section. This type of relationship is called a *referential constraint* because a foreign key in one table *references* the primary key in another table.

The customer and orders Tables

The **customer** table contains a **customer_num** column that holds a number that identifies a customer. The **orders** table also contains a **customer_num** column that stores the number of the customer who placed a particular order. In the **orders** table, the **customer_num** column is a foreign key that references the **customer_num** column in the **customer** table.

The orders and items Tables

The **orders** and **items** tables are linked by an **order_num** column that contains an identification number for each order. If an order includes several items, the same order number will be in several rows of the **items** table. In the **items** table, the **order_num** column is a foreign key that references the **order_num** column in the **orders** table.

The items and stock Tables

The **items** table and the **stock** table are joined by three columns: the **stock_num** column, which stores a stock number for an item, the **manu_code** column, which stores a code that identifies the manufacturer, and the **units** column, which identifies the types of unit in which the item can be ordered. You need the stock number, the manufacturer code, and the units to uniquely identify an item. The same stock number and manufacturer code can be in more than one row of the **items** table, if the same item belongs to separate orders. In the **items** table, the **stock_num**, **manu_code**, and **unit** columns are foreign keys that reference the **stock_num**, **manu_code**, and **unit** columns in the **stock** table.

The stock and catalog Tables

The **stock** table and **catalog** table are joined by three columns: the **stock_num** column, which stores a stock number for an item, the **manu_code** column, which stores a code that identifies the manufacturer, and the **unit** column, which identifies the type of units in which the item can be ordered. You need all three columns to uniquely identify an item. In the **catalog** table, the **stock_num**, **manu_code**, and **unit** columns are foreign keys that reference the **stock_num**, **manu_code**, and **unit** columns in the **stock** table.

The stock and manufact Tables

The **stock** table and the **manufact** table are joined by the **manu_code** column. The same manufacturer code can be in more than one row of the **stock** table if the manufacturer produces more than one piece of equipment. In the **stock** table, the **manu_code** column is a foreign key that references the **manu_code** column in the **manufact** table.

The cust_calls and customer Tables

The **cust_calls** table and the **customer** table are joined by the **customer_num** column. The same customer number can be in more than one row of the **cust_calls** table if the customer calls the distributor more than once with a problem or question. In the **cust_calls** table, the **customer_num** column is a foreign key that references the **customer_num** column in the **customer** table.

The call_type and cust_calls Tables

The **call_type** and **cust_calls** tables are joined by the **call_code** column. The same call code can be in more than one row of the **cust_calls** table, because many customers can have the same type of problem. In the **cust_calls** table, the **call_code** column is a foreign key that references the **call_code** column in the **call_type** table.

The state and customer Tables

The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer**

table. If several customers live in the same state, the same state code will be in several rows of the table. In the **customer** table, the **state** column is a foreign key that references the **code** column in the **state** table.

The customer and location Tables

In the **customer** table, the **customer_loc** column is a foreign key that references the **location_id** of the **location** table. The **customer_loc** and **location_id** columns each uniquely identify the customer location.

The manufact and location Tables

The **manu_loc** column in the **manufact** table is a foreign key that references the **location_id** column, which is the primary key in the **location** table. Both **manu_loc** and **location_id** uniquely identify the manufacturer location.

The state and location_us Tables

The **state** and **location_us** tables are joined by the column that contains the state code. The **state_code** column in the **location_us** table is a foreign key that references the **code** column in the **state** table.

The sales_rep and region Tables

The **region_num** column is the primary key in the **region** table. It is a system-generated region number. The **region_num** column in the **sales_rep** table is a foreign key that references and joins the **region_num** column in the region table.

The region and location Tables

The **region_loc** column in the **region** table identifies the regional office location. It is a foreign key that references the **location_id** column in the **location** table, which is a unique identifier for location.

The stock and stock_discount Tables

The **stock** table and the **stock_discount** table are joined by three columns: **stock_num**, **manu_code**, and **unit.** These columns form the primary key for the **stock** table. The **stock_discount** table has no primary key and references the **stock** table.

The stock and units Tables

The **unit_name** column of the **units** table is a primary key that identifies the kinds of units that can be ordered, such as case, pair, box, and so on. The **unit** column of the **stock** table joins the **unit_name** column of the **units** table

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.

Tip: The information center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features by using the keyboard instead of the mouse.

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.

You can view the publications in Adobe Portable Document Format (PDF) by using the Adobe Acrobat Reader.

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the IBM commitment to accessibility.

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 following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer 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 only 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.

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 (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.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy,

modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Printed in USA

SC27-3621-01

Spine information:

IBM Informix **Version 11.50**

IBM Informix Guide to SQL: Reference