HCL Informix 14.10

Designing databases

Contents

Chapter 1. Designing databases

The first step in creating a relational database is to construct a data model, which is a precise, complete definition of the data you want to store. After you prepare your data model, you must implement it as a database and tables. To implement your data model, you first select a data type for each column and then you create a database and tables and populate the tables with data. You can also implement fragmentation strategies and control access to your data.

Additional resources

> Database concepts Describes fundamental database concepts.
> Dimensional databases Design, implement, and manage dimensional databases.
> User-Defined Routines and Data Types Developer's Guide Define new data types, create new casts, extend operator classes for secondary-access methods, write opaque data types, and create and register routines.

Designing and Implementing a Database

The *Informix® Database Design and Implementation Guide* provides information to help you design, implement, and manage your Informix® databases. It includes data models that illustrate different approaches to database design and shows you how to use structured query language (SQL) to implement and manage your databases.

This publication is written for the following users:

- Database administrators
- Database server administrators
- Database-application programmers

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

This publication is one of several publications that contain information about the Informix® implementation of SQL. The *Informix® Guide to SQL: Tutorial* shows how to use basic and advanced SQL and Stored Procedure Language (SPL) routines to access and manipulate the data in your databases. The *Informix® Guide to SQL: Syntax* contains all the syntax descriptions for SQL and SPL. The *Informix® Guide to SQL: Reference* provides reference information for aspects of SQL other than the language statements.

Basics of database design and implementation

Plan a database

This chapter describes several issues that a database administrator (DBA) must understand to effectively plan for a database. It contains information about choosing a data model for your database, using ANSI-compliant databases, and using a customized language environment for your database.

Select a data model for your database

Before you create a database with the Informix® product, you must decide what type of data model you want to use to design your database. This manual describes the following database models:

Relational data model

This data model typifies database design for online transaction processing (OLTP). The purpose of OLTP is to process a large number of small transactions without losing any of them. An OLTP database is designed to handle the day-to-day requirements of a business, and database performance is tuned for those requirements. Basics of database design and implementation on page 1 of this manual, describes how to build and implement a relational data model for OLTP. Managing databases on page 49, contains information about how to manage your databases.

Object-relational data model

supports object-relational databases that employ basic relational design principles, but include features such as extended data types, user-defined routines, user-defined casts, and user-defined aggregates to extend the functionality of relational databases. Object-relational databases on page 96 of this manual, contains information about how to use the extensible features of Informix® to extend the kinds of data you can store in your database and to provide greater flexibility in how you organize and access your data.

Dimensional data model

This data model is typically used to build data marts, which are a type of data warehouse. In a data-warehousing environment, databases are optimized for data retrieval and analysis. This type of informational processing is known as online analytical processing (OLAP) or decision-support processing.

The remainder of this chapter describes the implications of these decisions and summarizes how the decisions that you make affect your database.

Use ANSI-compliant databases

You create an ANSI-compliant database when you use the MODE ANSI keywords in the CREATE DATABASE statement. However, creating an ANSI-compliant database does not ensure that this database remains ANSI-compliant. If you take a non-ANSI action (such as CREATE INDEX) on an ANSI database, you will receive a warning, but the application program does not forbid the action.

You might want to create an ANSI-compliant database for the following reasons:

- Privileges and access to objects

  ANSI rules govern privileges and access to objects such as tables and synonyms.
- Name isolation

  The ANSI table-naming scheme allows different users to create tables in a database without name conflicts.
- Transaction isolation

- Data recovery

  ANSI-compliant databases enforce unbuffered logging and implicit transactions for .

You can use the same SQL statements with both ANSI-compliant databases and non-ANSI-compliant databases.

Differences between ANSI-compliant and non-ANSI-compliant databases

Databases that you designate as ANSI-compliant and databases that are not ANSI-compliant behave differently in the following areas:

Transactions

A *transaction* is a collection of SQL statements that are treated as a single unit of work. All the SQL statements that you issue in an ANSI-compliant database are automatically contained in transactions. With a database that is not ANSI-compliant, transaction processing is an option.

In a database that is not ANSI-compliant, a transaction is enclosed by a BEGIN WORK statement and a COMMIT WORK or a ROLLBACK WORK statement. However, in an ANSI-compliant database, the BEGIN WORK statement is unnecessary, because all statements are automatically contained in a transaction. You are required to indicate only the end of a transaction with a COMMIT WORK or ROLLBACK WORK statement.

For more information about transactions, see Implement a relational data model on page 39 and the *Informix® Guide to SQL: Tutorial*.

Transaction logging

ANSI-compliant databases run with unbuffered transaction logging. In an ANSI-compliant database, you cannot change the logging mode to buffered logging, and you cannot turn logging off.

Databases of that are not ANSI-compliant can run with either buffered logging or unbuffered logging. Unbuffered logging provides more comprehensive data recovery, but buffered logging provides better performance.

For more information, see the description of the CREATE DATABASE statement in the *Informix® Guide to SQL: Syntax*.

Owner naming

In an ANSI-compliant database, owner naming is enforced. When you supply an object name in an SQL statement, ANSI standards require that the name include the prefix *owner*, unless you are the owner of the object. The combination of *owner* and *name* must be unique in the database. If you are the owner of the object, the database server supplies your user name as the default.

Databases that are not ANSI-compliant do not enforce owner naming. For more information, see the Owner Name segment in the *Informix® Guide to SQL: Syntax*.

Privileges on objects

ANSI-compliant databases and non-ANSI-compliant databases differ as to which users are granted table-level privileges by default when a table in a database is created. ANSI standards specify that the database server grants only the table owner (and the DBA if they are not the same user) any table-level privileges. In a database that is not ANSI-compliant, however, privileges are granted to PUBLIC. In addition, the database server provides two table-level privileges, Alter and Index, that are not included in the ANSI standards.

To run a user-defined routine, you must have the Execute privilege for that routine. When you create an owner-privileged procedure for an ANSI-compliant database, only the owner of the user-defined routine has the Execute privilege. When you create an owner-privileged routine in a database that is not ANSI-compliant, the database server grants the Execute privilege to PUBLIC by default.

Setting the **NODEFDAC** environment variable to 'yes' causes a database that is not ANSI-compliant to emulate the behavior of an ANSI-compliant database in not granting privileges to PUBLIC automatically when a user creates a table or an owner-privileged routine. For more information about privileges, see Grant and limit access to your database on page 60 and the description of the GRANT statement in the *Informix® Guide to SQL: Syntax*.

Default isolation level

The database isolation level specifies the degree to which your program is isolated from the concurrent actions of other programs. The default isolation level for all ANSI-compliant databases is Repeatable Read. The default isolation level for non-ANSI-compliant databases that support transaction logging is Committed Read. The default isolation level for non-ANSI-compliant databases that do not use transaction logging is Uncommitted Read. For information about isolation levels, see the *Informix® Guide to SQL: Tutorial* and the description of the SET TRANSACTION and SET ISOLATION statements in the *Informix® Guide to SQL: Syntax*.

Character data types

If a database is not ANSI-compliant, you get no error if a character field (CHAR, CHARACTER, LVARCHAR, NCHAR, NVARCHAR, VARCHAR, CHARACTER VARYING) receives a string that is longer than the specified length of the field. The database server truncates the extra characters without resulting in an error message. Thus the semantic integrity of data for a CHAR(n) column or variable is not enforced when the value inserted or updated exceeds n bytes.

In an ANSI-compliant database, you get an error if any character field (CHAR, CHARACTER, LVARCHAR, NCHAR, NVARCHAR, VARCHAR, CHARACTER VARYING) receives a string that is longer than the specified width of the field.

DECIMAL data type

If a database is not ANSI-compliant, a DECIMAL data type that you declare with a precision but no scale can store floating point values of the specified precision. If you specify neither precision nor scale, the default precision is 16.

In an ANSI-compliant database, all DECIMAL values are fixed-point and must be declared with an explicit precision. If you specify no scale for the DECIMAL data type, the scale = 0, and only integer values can be stored.

Escape characters

In an ANSI-compliant database, escape characters can only escape the special significance of the percent (%) and underscore (_) characters. You can also use an escape character to escape itself. For more information about escape characters, see the Condition segment in the *Informix® Guide to SQL: Syntax*.

Cursor behavior

If a database is not ANSI-compliant, you must use the FOR UPDATE keywords when you declare an update cursor for a SELECT statement. The SELECT statement must also meet the following conditions:

- It selects from a single table.
- It does not include any aggregate functions.
- It does not include the DISTINCT, GROUP BY, INTO TEMP, ORDER BY, UNION, or UNIQUE clauses and keywords.

In ANSI-compliant databases, the FOR UPDATE keywords are implicit when you declare a cursor, and all cursors that meet the restrictions that the preceding list describes are potentially update cursors. You can specify that a cursor is read-only with the FOR READ ONLY keywords on the DECLARE statement.

For more information, see the description of the DECLARE statement in the *Informix® Guide to SQL: Syntax*.

SQLCODE field of the SQL communications area

If no rows satisfy the search criteria of a DELETE, an INSERT INTO *tablename* SELECT, a SELECT...INTO TEMP, or an UPDATE statement, the database server sets SQLCODE to 100 if the database is ANSI-compliant and 0 if the database is not ANSI-compliant.

For more information, see the descriptions of SQLCODE in the *Informix® Guide to SQL: Tutorial*.

Synonym behavior

Synonyms are always private in an ANSI-compliant database. If you attempt to create a public synonym or use the PRIVATE keyword to designate a private synonym in an ANSI-compliant database, you receive an error.

For more information, see the description of the CREATE SYNONYM statement in the *Informix® Guide to SQL: Syntax*.

Determine if an existing database is ANSI-compliant

The following list describes two methods to determine whether a database is ANSI-compliant:

- From the **sysmaster** database you can execute the following statement:

  ```
  SELECT name,is_ansi FROM sysmaster:sysdatabases
  ```

  The query returns the value 1 for ANSI-compliant databases and 0 for non-ANSI-compliant databases for each database on your database server.
- If you are using an SQL API such as Informix® ESQL/C, you can test the SQL Communications Area (SQLCA). Specifically, the third element in the SQLCAWARN structure contains a `w` immediately after you open an ANSI-compliant database with the DATABASE or CONNECT statement. For information about SQLCA, see the *Informix® Guide to SQL: Tutorial* or your SQL API manual.

Use a customized language environment for your database (GLS)

Global Language Support (GLS) permits you to use different locales. A GLS locale is an environment that has defined conventions for a particular language or culture.

By default, Informix® products use the U.S. English ASCII code set and perform in the U.S. English environment with ASCII collation order. Set your environment to accommodate a nondefault locale if you plan to use any of the following functions:

- Non-ASCII characters in the data
- Non-ASCII characters in user-specified object names
- Conformity with the sorting and collation order of a non-default code set
- Culture-specific collation and sorting orders, such as those used in dictionaries or phone books

supports the UTF-8 Unicode locale.

Unlike other locales, UTF-8 enables a single database to store character strings from two or more natural languages that use dissimilar code sets.

For descriptions of GLS environment variables and for detailed information about how to implement non-default locales, see the *Informix® GLS User's Guide*.

Build a relational data model

The first step in creating a relational database is to construct a data model: a precise, complete definition of the data you want to store. This chapter provides an overview of one way to model the data. For information about defining column-specific properties of a data model, see Select data types on page 21. To learn how to implement the data model that this chapter describes, see Implement a relational data model on page 39.

To understand the material in this chapter, a basic understanding of SQL and relational database theory are necessary.

Build a data model

You already have some idea about the type of data in your database and how that data must be organized. This information is the beginning of a data model. Building a data model with formal notation has the following advantages:

- You think through the data model completely.

  A mental model often contains unexamined assumptions; when you formalize the design, you discover these assumptions.
- The design is easier to communicate to other people.

  A formal statement makes the model explicit, so that others can return comments and suggestions in the same form.

Overview of the entity-relationship data model

More than one formal method for data modeling exists. Most methods force you to be thorough and precise. If you know a method, by all means use it.

This chapter presents a summary of the entity-relationship (E-R) data model. The E-R data-modeling method follows these steps:

1. Identify and define the principal data objects (entities, relationships, and attributes).
2. Diagram the data objects using the E-R approach.
3. Translate the E-R data objects into relational constructs.
4. Resolve the logical data model.
5. Normalize the logical data model.

Steps 1 through 5 are explained in this chapter. Implement a relational data model on page 39 contains information about the final step of converting your logical data model to a physical schema.

The end product of data modeling is a fully-defined database design encoded in a diagram similar to Figure 11: The data model of a personal telephone directory on page 20, which shows the final set of tables for a personal telephone directory. The personal telephone directory is an example developed in this chapter. It is used rather than the demonstration database because it is small enough to be developed completely in one chapter but large enough to show the entire method.

Identify and define principal data objects

To create a data model, you first identify and define the principal data objects: entities, relationships, and attributes.

Discover entities

An *entity* is a principal data object that is of significant interest to the user. It is usually a person, place, thing, or event to be recorded in the database. If the data model were a language, entities would be nouns. The demonstration database provided with your software contains the following entities:

- *customer*
- *orders*
- *items*
- *stock*
- *catalog*
- *cust_calls*
- *call_type*
- *manufact*
- *state*

List of entities

When the list of entities seems complete, check the list to make sure that each entity has the following qualities:

- It is significant.

  List only entities that are important to your database users and that are worth the trouble and expense of computer tabulation.
- It is generic.

  List only types of things, not individual instances. For instance, *symphony* might be an entity, but *Beethoven's Fifth* would be an entity instance or entity occurrence.
- It is fundamental.

  List only entities that exist independently and do not require anything else to explain them. Anything you might call a trait, a feature, or a description is not an entity. For example, a *part number* is a feature of the fundamental entity called *part*. Also, do not list things that you can derive from other entities; for example, avoid any sum, average, or other quantity that you can calculate in a SELECT expression.
- It is unitary.

  Be sure that each entity you name represents a single class. It cannot be separated into subcategories, each with its own features. In the telephone directory example in , the telephone number, an apparently simple entity, actually consists of three categories, each with different features.

These choices are neither simple nor automatic. To discover the best choice of entities, you must think carefully about the nature of the data you want to store. Of course, that is exactly the point of a formal data model. The following section describes the telephone directory example in detail.

Telephone directory example

Suppose that you create a database for a personal telephone directory. The database model must record the names, addresses, and telephone numbers of people and organizations that the user requires.

First define the entities. Look carefully at a page from a telephone directory to identify the entities that it contains. The following figure shows a sample page from a telephone directory.

Figure 1. Partial page from a telephone directory
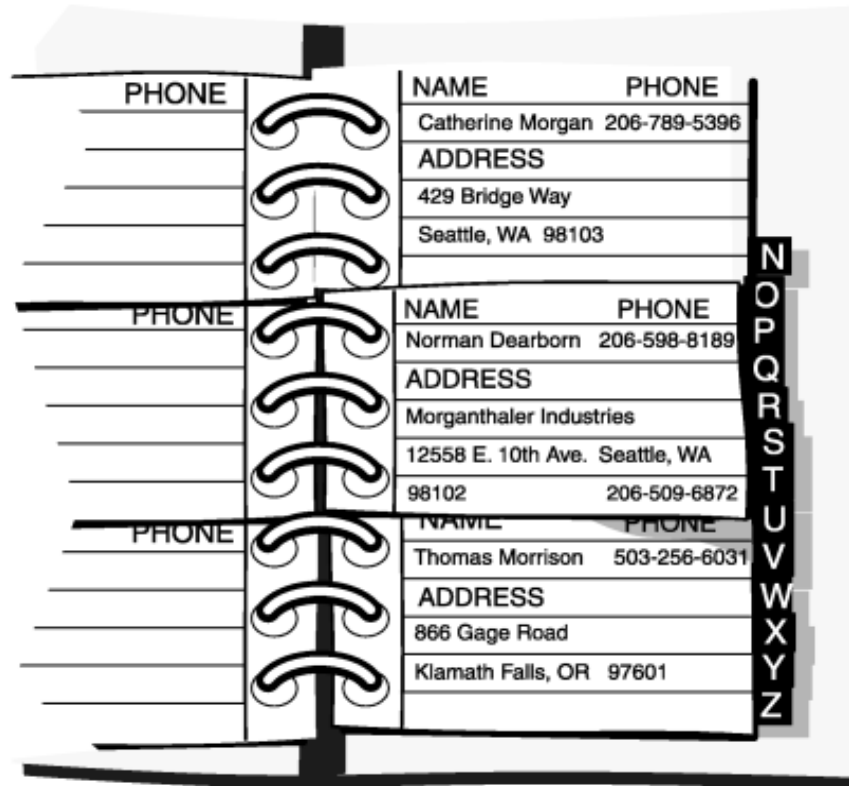
The physical form of the existing data can be misleading. Do not let the layout of pages and entries in the telephone directory mislead you into trying to specify an entity that represents one entry in the book: an alphabetized record with fields for name, number, and address. You want to model the data, not the medium.

Generic and significant entities

At first glance, the entities that are recorded in a telephone directory include the following items:

- Names (of persons and organizations)
- Addresses
- Telephone numbers

Do these entities meet the earlier criteria? They are clearly significant to the model and are generic.

Fundamental entities

A good test is to ask if an entity can vary in number independently of any other entity. A telephone directory sometimes lists people who have no number or current address (people who move or change jobs) and also can list both addresses and

numbers that more than one person uses. All three of these entities can vary in number independently; this fact strongly suggests that they are fundamental, not dependent.

Unitary entities

Names can be split into personal names and corporate names. You decide that all names should have the same features in this model; that is, you do not plan to record different information about a company than you would record about a person. Likewise, you decide that only one kind of address exists; you are not required to treat home addresses differently from business addresses.

However, you also realize that more than one kind of telephone number exists. Voice numbers are answered by a person, fax numbers connect to a fax machine, and modem numbers connect to a computer. You decide that you want to record different information about each kind of number, so these three types are different entities.

For the personal telephone directory example, you decide that you want to keep track of the following entities:

- Name
- Address (mailing)
- Telephone number (voice)
- Telephone number (fax)
- Telephone number (modem)

Diagram entities

Later in this chapter you can learn how to use the E-R diagrams. For now, create a separate, rectangular box for each entity in the telephone directory example, as the following figure shows. Diagram data objects on page 10 shows how to put the entities together with relationships.

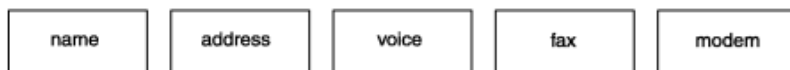Figure 2. Entities in the personal telephone directory example


Diagram data objects

Now you know and understand the entities and relationships in your database, which is the most important part of the relational-database design process. After you determine the entities and relationships, a method that displays your thought process during database design might be helpful.

Most data-modeling methods provide some way to graphically display the entities and relationships. Informix® documentation uses the E-R diagram approach that C. R. Bachman originally developed. E-R diagrams serve the following purposes. They:

- Model the informational requirements of an organization
- Identify entities and their relationships
- Provide a starting point for data definition (data-flow diagrams)
- Provide an excellent source of documentation for application developers and both database and system administrators
- Create a logical design of the database that can be translated into a physical schema

Several different styles of E-R diagrams exist. If you already have a style that you prefer, use it. Figure 3: Symbols of an Entity-Relationship diagram on page 11 shows a sample E-R diagram.

Figure 3. Symbols of an Entity-Relationship diagram
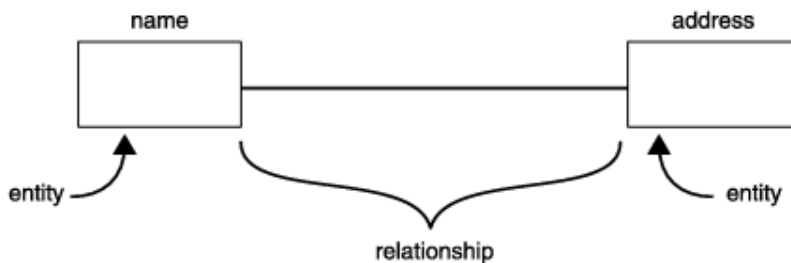
In an E-R diagram, a box represents an entity. A line represents the relationships that connect the entities. In addition, Figure 4: The parts of a relationship in an Entity-Relationship diagram on page 11 shows how you use graphical items to display the following features of relationships:

- A circle across a relationship link indicates *optionality* in the relationship (zero instances can occur).
- A small bar across a relationship link indicates that exactly one instance of the entity is associated with another entity (consider the bar to be a 1).
- The crow's-feet represent many in the relationship.

  Figure 4. The parts of a relationship in an Entity-Relationship diagram
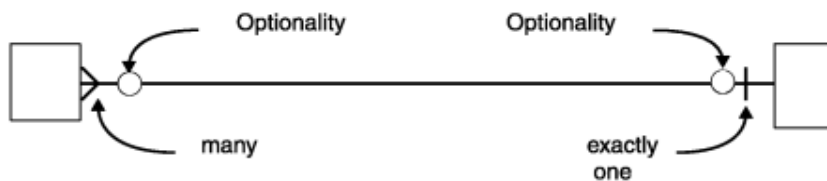
How to read E-R diagrams

You read the diagrams first from left to right and then from right to left. In the case of the name-address relationship in following figure, you read the relationships as follows: names can be associated with zero or exactly one address; addresses can be associated with zero, one, or many names.

Figure 5. Reading an entity-relationship diagram
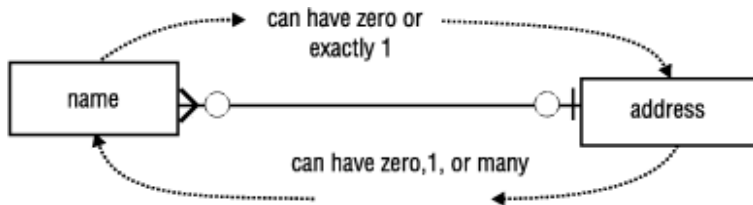
Telephone directory example

The following figure shows the telephone directory example and includes the entities, relationships, and attributes. This diagram includes the relationships that you establish with the matrix. After you study the diagram symbols, compare the E-R diagram in the following figure with the matrix in . Verify for yourself that the relationships are the same in both figures.

A matrix such as is a useful tool when you first design your model, because when you fill it out, you are forced to think of every possible relationship. However, the same relationships appear in a diagram such as the following figure, and this type of diagram might be easier to read when you review an existing model.

Figure 6. Preliminary entity-relationship diagram of the telephone directory example
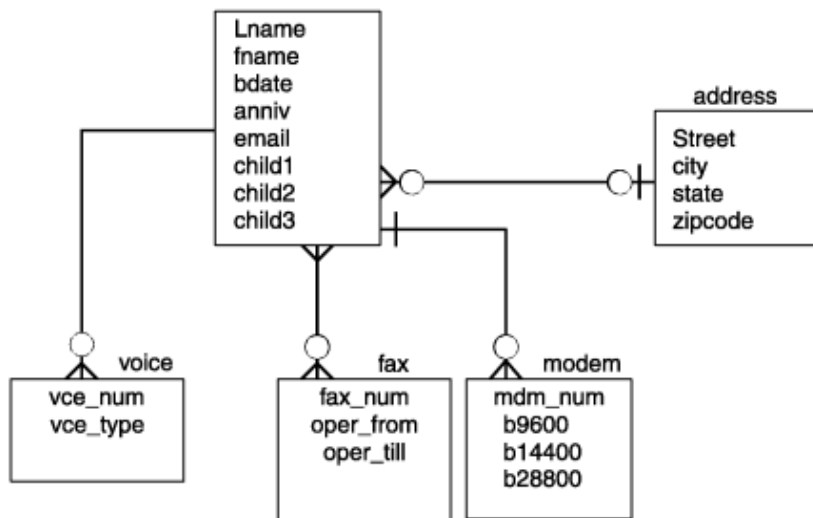
After the diagram is complete

The rest of this chapter describes how to perform the following tasks:

- Translate the entities, relationships, and attributes into relational constructs.
- Resolve the E-R data model.
- Normalize the E-R data model.

Implement a relational data model on page 39 shows you how to create a database from the E-R data model.

Translate E-R data objects into relational constructs

All the data objects you have learned about so far (entities, relationships, attributes, and entity occurrences) translate into SQL tables, joins between tables, columns, and rows. The tables, columns, and rows of your database must fit the rules found in Define tables, rows, and columns on page 13.

Before you normalize your data objects, check that they fit these rules. To normalize your data objects, analyze the dependencies between the entities, relationships, and attributes. Normalization is explained in Normalize a data model on page 18.

After you normalize the data model, you can use SQL statements to create a database that is based on your data model. Implement a relational data model on page 39 describes how to create a database and provides the database schema for the telephone directory example.

Each entity that you select is represented as a table in the model. The table stands for the entity as an abstract concept, and each row represents a specific, individual occurrence of the entity. In addition, each attribute of an entity is represented by a column in the table.

The following ideas are fundamental to most relational data-model methods, including the E-R data model. Follow these rules while you design your data model to save time and effort when you normalize your model.

Define tables, rows, and columns

You are already familiar with the idea of a *table* that is composed of *rows* and *columns*. But you must respect the following rules when you define the tables of a formal data model:

- Rows must stand alone.

  Each row of a table is independent and does not depend on any other row of the same table. As a consequence, the order of the rows in a table is not significant in the model. The model should still be correct even if all the rows of a table are shuffled into random order.

  After the database is implemented, you can tell the database server to store rows in a certain order for the sake of efficiency, but that order does not affect the model.
- Rows must be unique.

  In every row, some column must contain a unique value. If no single column has this property, the values of some group of columns taken as a whole must be different in every row.
- Columns must stand alone.

  The order of columns within a table has no meaning in the model. The model should still be correct even if the columns are rearranged.

  After the database is implemented, programs and stored queries that use an asterisk to mean *all columns* are dependent on the final order of columns, but that order does not affect the model.
- Column values must be unitary.

A column can contain only single values, never lists or repeating groups. Composite values must be separated into individual columns. For example, if you decide to treat a person's first and last names as separate values, as the examples in this chapter show, the names must be in separate columns, not in a single **name** column.

- Each column must have a unique name.

  Two columns within the same table cannot share the same name. However, you can have columns that contain similar information. For example, the **name** table in the telephone directory example contains columns for children's names. You can name each column, child1, child2, and so on.

- Each column must contain data of a single type.

  A column must contain information of the same data type. For example, a column that is identified as an integer must contain only numeric information, not characters from a name.

If your previous experience is only with data organized as arrays or sequential files, these rules might seem unnatural. However, relational database theory shows that you can represent all types of data with only tables, rows, and columns that follow these rules. With a little practice, these rules become automatic.

Place constraints on columns

When you define your table and columns with the CREATE TABLE statement, you constrain each column. These constraints specify whether you want the column to contain characters or numbers, the form that you want dates to use, and other constraints. A *domain* describes the constraints when it identifies the set of valid values that attributes can assume.

Domain characteristics

You define the domain characteristics of columns when you create a table. A column can contain the following domain characteristics:

- Data type (INTEGER, CHAR, DATE, and so on)
- Format (for example, yy/mm/dd)
- Range (for example, 1,000 to 5,400)
- Meaning (for example, serial number)
- Allowable values (for example, only grades S or U)
- Uniqueness
- Null support
- Default value
- Referential constraints

For information about how to define domains, see Select data types on page 21. For information about how to create your tables and database, see Implement a relational data model on page 39.

Determine keys for tables

The columns of a table are either *key* columns or *descriptor* columns. A key column is one that uniquely identifies a particular row in the table. For example, a social security number is unique for each employee. A descriptor column specifies the nonunique characteristics of a particular row in the table. For example, two employees can have the same first name, Sue. The first name Sue is a nonunique characteristic of an employee. The main types of keys in a table are primary keys and foreign keys.

You designate primary and foreign keys when you create your tables. Primary and foreign keys are used to relate tables physically. Your next task is to specify a primary key for each table. That is, you must identify some quantifiable characteristic of the table that distinguishes each row from every other row.

Primary keys

The *primary key* of a table is the column whose values are different in every row. Because they are different, they make each row unique. If no one such column exists, the primary key is a composite of two or more columns whose values, taken together, are different in every row.

Every table in the model must have a primary key. This rule follows automatically from the rule that all rows must be unique. If necessary, the primary key is composed of all the columns taken together. You shouldn't use long character strings as primary keys.

For efficiency, the primary key should be one of the following types:

- Numeric (INT or SMALLINT)
- Serial (BIGSERIAL, SERIAL, or SERIAL8)
- A short character string (as used for codes).

NULL values are never allowed in a primary-key column. NULL values are not comparable; that is, they cannot be said to be alike or different. Hence, they cannot make a row unique from other rows. If a column permits NULL values, it cannot be part of a primary key. When you define a PRIMARY KEY constraint, the database server also silently creates a NOT NULL constraint on the same column, or on the same set of columns that make up the primary key.

Some entities have ready-made primary keys such as catalog codes or identity numbers, which are defined outside the model. Sometimes more than one column or group of columns can be used as the primary key. All columns or groups that qualify to be primary keys are called *candidate keys*. All candidate keys are worth noting because their property of uniqueness makes them predictable in a SELECT operation.

Composite keys

Some entities lack features that are reliably unique. Different people can have identical names; different books can have identical titles. You can usually find a composite of attributes that work as a primary key. For example, people rarely have identical names and identical addresses, and different books rarely have identical titles, authors, and publication dates.

System-assigned keys

A system-assigned primary key is usually preferable to a composite key. A system-assigned key is a number or code that is attached to each instance of an entity when the entity is first entered into the database. The easiest system-assigned keys to implement are serial numbers because the database server can generate them automatically. Informix® database servers offer the SERIAL, SERIAL8, and BIGSERIAL data types for serial numbers. However, the people who use the database might not like a plain numeric code. Other codes can be based on actual data; for example, an employee identification code can be based on a person's initials combined with the digits of the date that they were hired. In the telephone directory example, a system-assigned primary key is used for the **name** table.

Foreign keys (join columns)

A *foreign key* is a column or group of columns in one table that contains values that match the *primary key* in another table. Foreign keys are used to join tables. The following figure shows the primary and foreign keys of the **customer** and **orders** tables from the demonstration database.

Figure 7. Primary and foreign keys in the customer-order relationships

Primary key   Foreign key
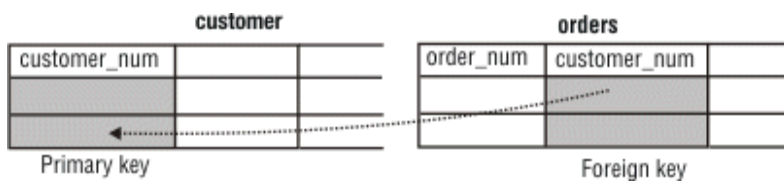
> **ⓘ Tip:** For ease in maintaining and using your tables, it is important to use names for the primary and foreign keys so that the relationship is readily apparent. In Figure 7: Primary and foreign keys in the customer-order relationships on page 16, both the primary and foreign key columns have the same name, **customer_num**. Alternatively, you might name the columns in Figure 7: Primary and foreign keys in the customer-order relationships on page 16 **customer_custnum** and **orders_custnum**, so that each column has a distinct name.

Foreign keys are noted wherever they appear in the model because their presence can restrict your ability to delete rows from tables. Before you can delete a row safely, either you must delete all rows that refer to it through foreign keys, or you must define the relationship with special syntax that allows you to delete rows from primary-key and foreign-key columns with a single delete command. The database server does not allow deletions that violate referential integrity.

To preserve referential integrity, delete all foreign-key rows before you delete the primary key to which they refer. If you impose referential constraints on your database, the database server does not permit you to delete primary keys with matching foreign keys. It also does not permit you to add a foreign-key value that does not reference an existing primary-key value. For more information about referential integrity, see the *Informix® Guide to SQL: Tutorial*.

Add keys to the telephone directory diagram

The following figure shows the initial choices of primary and foreign keys. This diagram reflects some important decisions.

For the **name** table, the primary key **rec_num** is selected. The data type for **rec_num** is SERIAL. The values for **rec_num** are system generated. If you look at the other columns (or attributes) in the **name** table, you see that the data types that are associated with the columns are mostly character-based. None of these columns alone is a good candidate for a primary key. If you combine elements of the table into a composite key, you create a cumbersome key. The SERIAL data type gives you a key that you can also use to join other tables to the **name** table.

The **voice**, **fax**, **modem**, and **address** tables are each joined to **name** through the **rec_num** key.

For the **voice**, **fax**, and **modem** tables the telephone numbers are used as primary keys. The **address** table contains a special column (**id_num**) that serves no other purpose than to act as a primary key. This is done because if **id_num** did not exist then all of the other columns would have to be used together as a composite key in order to guarantee that no duplicate primary keys existed. Using all of the columns as a primary key would be very inefficient and confusing.

Figure 8. Telephone directory diagram with primary and foreign keys added
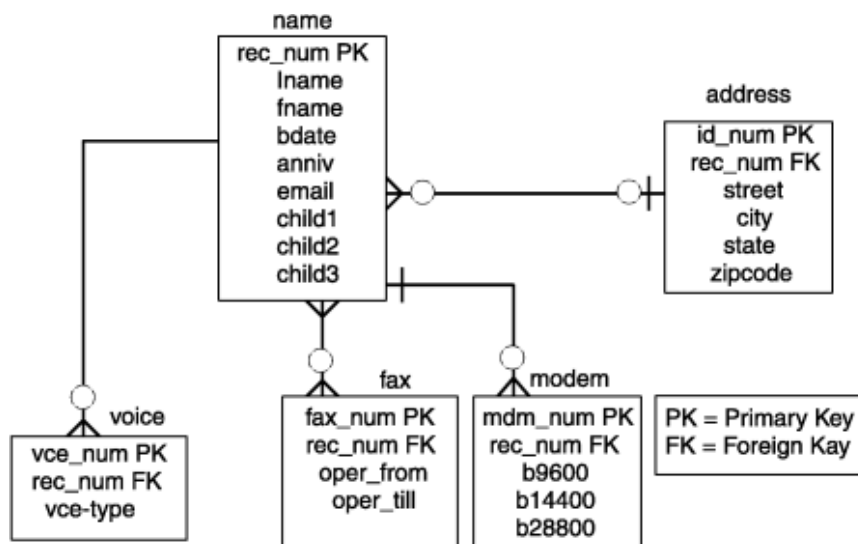
Resolve relationships

The aim of a good data model is to create a structure that provides the database server with quick access. To further refine the telephone directory data model, you can resolve the relationships and normalize the data model. This section addresses how and why to resolve your database relationships. Normalizing your data model is explained in .

Resolve other special relationships

You might encounter other special relationships that can hamper a smooth-running database. The following describes these relationships:

Complex relationship

A complex relationship is an association among three or more entities. All the entities must be present for the relationship to exist. To reduce this complexity, reclassify all complex relationships as an entity, related through binary relationships to each of the original entities.

Recursive relationship

A recursive relationship is an association between occurrences of the same entity type. These types of relationships do not occur often. Examples of recursive relationships are bills-of-materials (parts are composed of subparts) and organizational structures (employee manages other employees). You might not resolve recursive relationships. For an extended example of a recursive relationship, see the *Informix® Guide to SQL: Tutorial*.

Redundant relationship

A redundant relationship exists when two or more relationships represent the same concept. Redundant relationships add complexity to the data model and lead a developer to place attributes in the model incorrectly. Redundant relationships might appear as duplicated entries in your E-R diagram. For example, you might have two entities that contain the same attributes. To resolve a redundant relationship, review your data model. Do you have more than one entity that contains the same attributes? You might be required to add an entity to the model to resolve the redundancy. Your *Informix® Performance Guide* contains information about additional topics that are related to redundancy in a data model.

Normalize a data model

The telephone directory example in this chapter appears to be a good model. You can implement it at this point into a database, but this example might present problems later with application development and data-manipulation operations. *Normalization* is a formal approach that applies a set of rules to associate attributes with entities.

When you normalize your data model, you can achieve the following goals. You can:

- Produce greater flexibility in your design.
- Ensure that attributes are placed in the correct tables.
- Reduce data redundancy.
- Increase programmer effectiveness.
- Lower application maintenance costs.
- Maximize stability of the data structure.

Normalization consists of several steps to reduce the entities to more desirable physical properties. These steps are called normalization rules, also referred to as *normal forms*. Several normal forms exist; this chapter contains information about the first three normal forms. Each normal form constrains the data more than the last form. Because of this, you must achieve first normal form before you can achieve second normal form, and you must achieve second normal form before you can achieve third normal form.

First normal form

An entity is in the first normal form if it contains no repeating groups. In relational terms, a table is in the first normal form if it contains no repeating columns. Repeating columns make your data less flexible, waste disk space, and make it more difficult to search for data. In the following telephone directory example, the **name** table contains repeating columns, child1, child2, and child3.

Figure 9. Name entity before normalization
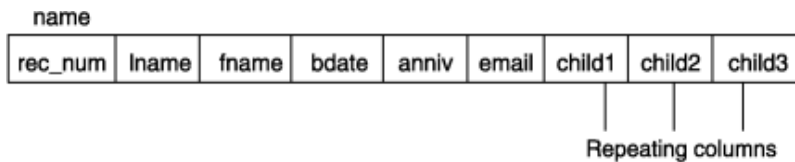
You can see some problems in the current table. The table always reserves space on the disk for three child records, whether the person has children or not. The maximum number of children that you can record is three, but some of your acquaintances might have four or more children. To look for a particular child, you must search all three columns in every row.

To eliminate the repeating columns and bring the table to the first normal form, separate the table into two tables. Put the repeating columns into one of the tables. The association between the two tables is established with a primary-key and foreign-key combination. Because a child cannot exist without an association in the **name** table, you can reference the **name** table with a foreign key, **rec_num**.

Figure 10. First normal form reached for name entity
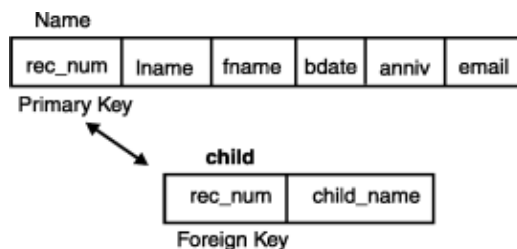
Now check the telephone directory structure in for groups that are not in the first normal form. The name-modem relationship is not in the first normal form because the columns **b9600**, **b14400**, and **b28800** are considered repeating columns. Add a new attribute called **b_type** to the **modem** table to contain occurrences of **b9600**, **b14400**, and **b28800**. The following diagram shows the data model normalized through the first normal form.

Figure 11. The data model of a personal telephone directory
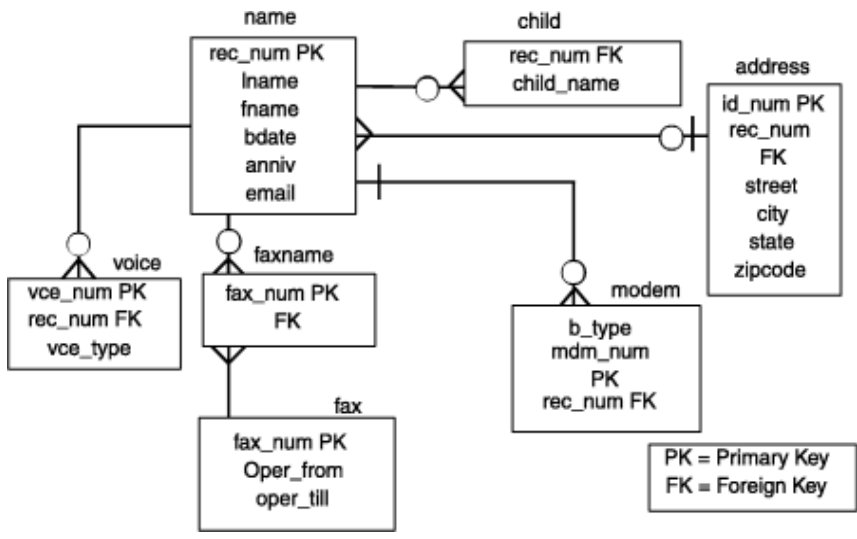
Second normal form

An entity is in the second normal form if all of its attributes depend on the whole (primary) key. In relational terms, every column in a table must be functionally dependent on the whole primary key of that table. Functional dependency indicates that a link exists between the values in two different columns.

If the value of an attribute depends on a column, the value of the attribute must change if the value in the column changes. The attribute is a function of the column. The following explanations make this more specific:

- If the table has a one-column primary key, the attribute must depend on that key.
- If the table has a composite primary key, the attribute must depend on the values in all its columns taken as a whole, not on one or some of them.
- If the attribute also depends on other columns, they must be columns of a candidate key; that is, columns that are unique in every row.

If you do not convert your model to the second normal form, you risk data redundancy and difficulty in changing data. To convert first-normal-form tables to second-normal-form tables, remove columns that are not dependent on the primary key.

Third normal form

An entity is in the third normal form if it is in the second normal form and all of its attributes are not transitively dependent on the primary key. *Transitive dependence* means that descriptor key attributes depend not only on the whole primary key, but also on other descriptor key attributes that, in turn, depend on the primary key. In SQL terms, the third normal form means that no column within a table is dependent on a descriptor column that, in turn, depends on the primary key.

To convert to third normal form, remove attributes that depend on other descriptor key attributes.

Summary of normalization rules

The following normal forms are explained in this section:

First normal form

A table is in the first normal form if it contains no repeating columns.

Second normal form

A table is in the second normal form if it is in the first normal form and contains only columns that are dependent on the whole (primary) key.

Third normal form

A table is in the third normal form if it is in the second normal form and contains only columns that are nontransitively dependent on the primary key.

When you follow these rules, the tables of the model are in the third normal form, according to E. F. Codd, the inventor of relational databases. When tables are not in the third normal form, either redundant data exists in the model, or problems exist when you attempt to update the tables.

If you cannot find a place for an attribute that observes these rules, you have probably made one of the following errors:

- The attribute is not well defined.
- The attribute is derived, not direct.
- The attribute is really an entity or a relationship.
- Some entity or relationship is missing from the model.

Select data types

After you prepare your data model, you must implement it as a database and tables. To implement your data model, you first define a domain, or set of data values, for every column. This chapter contains information about the decisions that you must make to define the column data types and constraints.

Implement a relational data model on page 39 contains information about how the second step uses the CREATE DATABASE and CREATE TABLE statements to implement the model and populate the tables with data.

Define the domains

To complete the data model that Build a relational data model on page 6 describes, you must define a domain for each column. The *domain* of a column describes the constraints and identifies the set of valid values that attributes (columns) can assume.

The purpose of a domain is to guard the *semantic integrity* of the data in the model; that is, to ensure that it reflects reality in a sensible way. The integrity of the data model is at risk if you can substitute a name for a telephone number or if you can enter a fraction where only integers are valid values.

To define a domain, specify the constraints that a data value must satisfy before it can be part of the domain. To specify a column domain, use the following constraints:

- Data types
- Default values
- Check constraints
- Referential constraints

Data types

The first constraint on any column is the one that is implicit in the data type for the column. When you select a data type, you constrain the column so that it contains only values that can be represented by that data type.

Each data type represents certain kinds of information and not others. The correct data type for a column is the one that represents all the data values that are appropriate for that column but contains as few as possible of the values that are not appropriate for it.

This chapter describes built-in data types.

For information about the extended data types that supports, see Create and use extended data types in Informix on .

Select a data type

Every column in a table must have a data type. The choice of data type is important for the following reasons:

- It establishes the set of valid data items that the column can store.
- It determines the kinds of operations that you can perform on the data.

  For example, you cannot apply aggregate functions, such as SUM, to columns that are defined on a character data type.
- It determines how much space each data item occupies on disk.

  The space required to accommodate data items is not as important for small tables as it is for tables with hundreds of thousands of rows. When a table reaches that many rows, the difference between a 4-byte and an 8-byte data type can be crucial.

The following figure shows a decision tree that summarizes the choices among built-in data types. The choices are explained in the following sections.
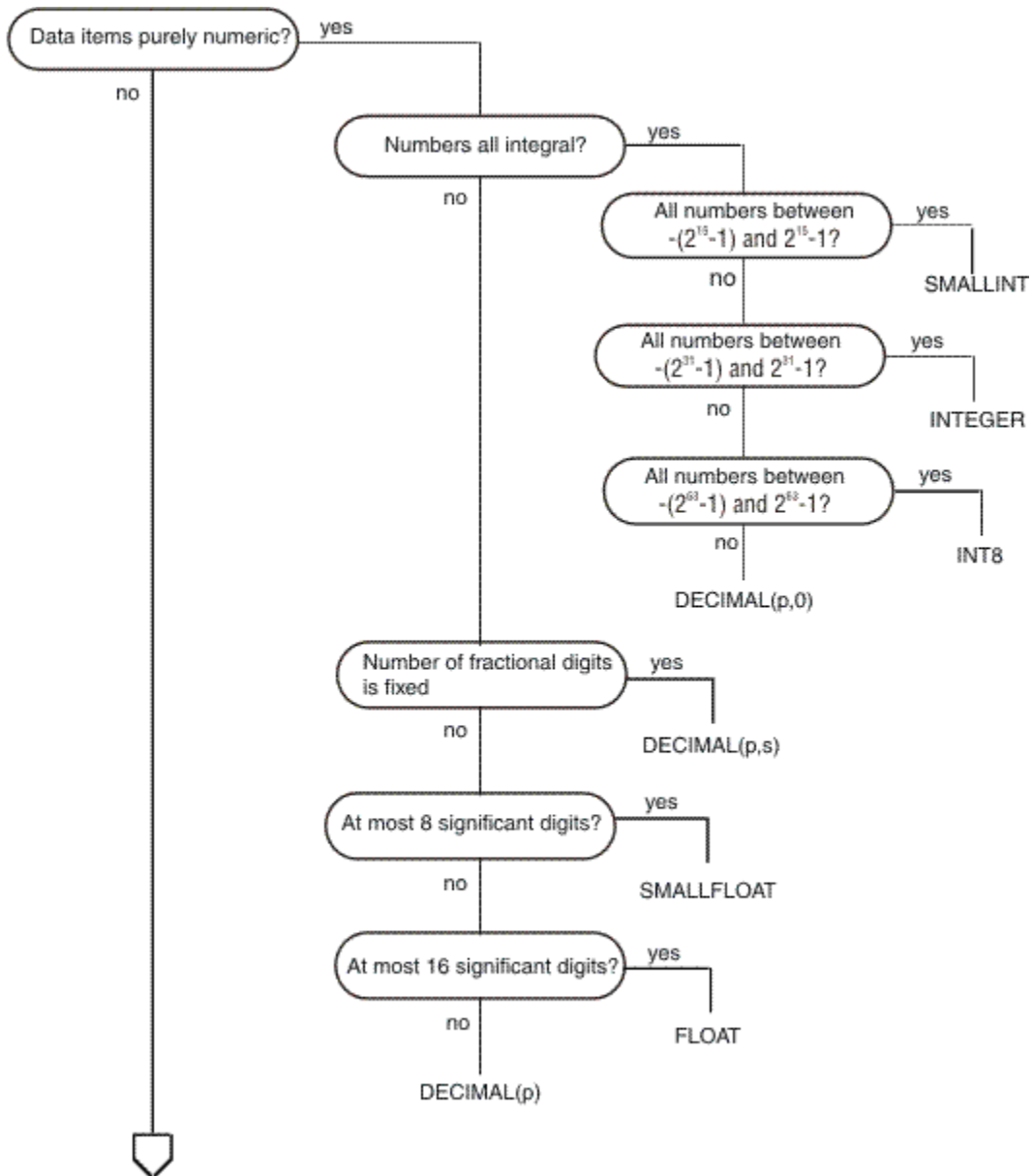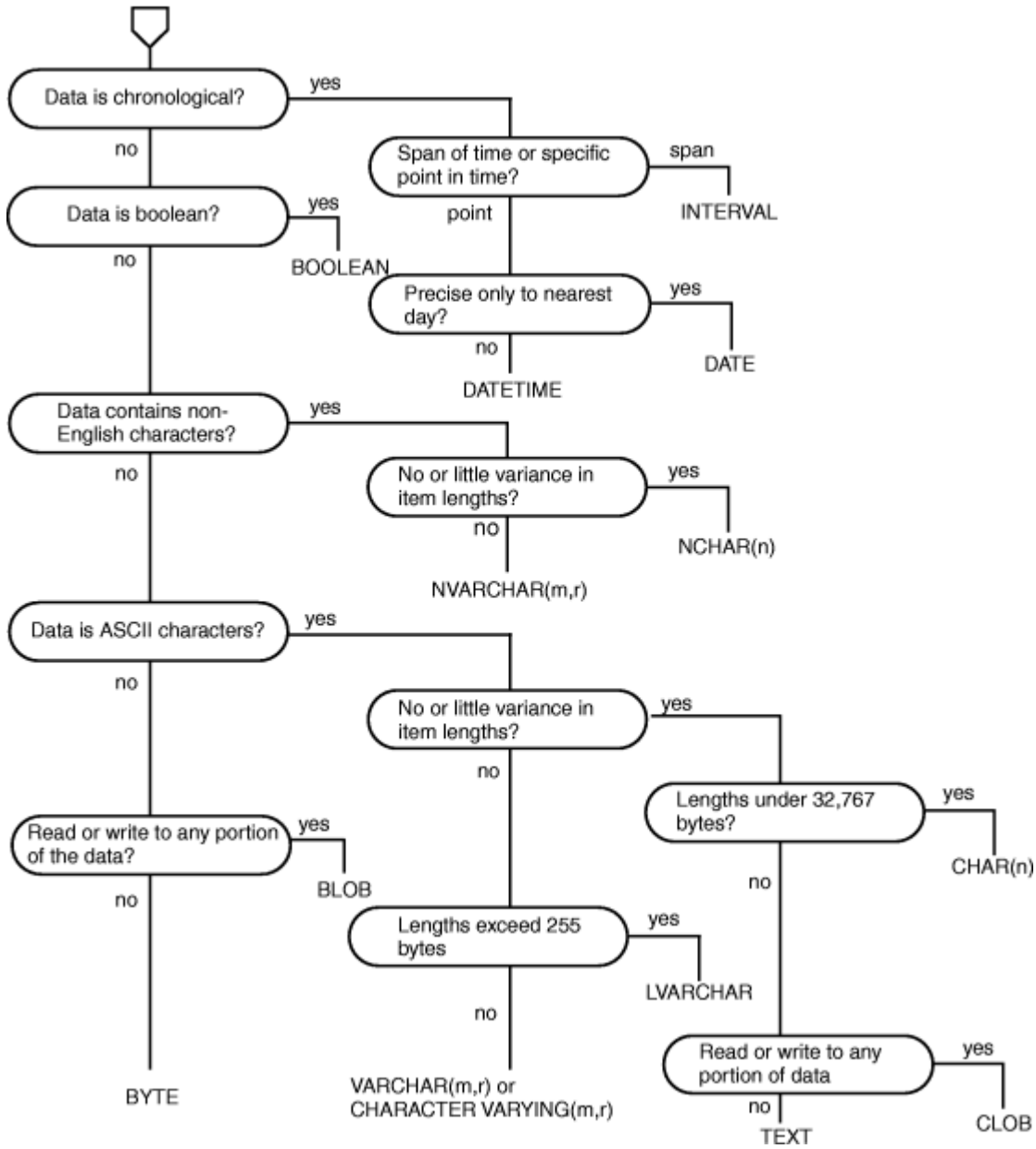
Figure 12. Select a data type

Data items purely numeric? — yes
no

Numbers all integral? — yes
no

All numbers between $-(2^{15}-1)$ and $2^{15}-1$? — yes → SMALLINT
no

All numbers between $-(2^{31}-1)$ and $2^{31}-1$? — yes → INTEGER
no

All numbers between $-(2^{63}-1)$ and $2^{63}-1$? — yes → INT8
no → DECIMAL(p,0)

Number of fractional digits is fixed — yes → DECIMAL(p,s)
no

At most 8 significant digits? — yes → SMALLFLOAT
no

At most 16 significant digits? — yes → FLOAT
no → DECIMAL(p)

Figure 13. Select a data type (continued)

Numeric types

Some numeric data types are best suited for counters and codes, some for engineering quantities, and some for money.

Counters and codes: BIGINT, INT8, INTEGER, and SMALLINT

The INTEGER and SMALLINT data types hold small whole numbers. They are suited for columns that contain counts, sequence numbers, numeric identity codes, or any range of whole numbers when you know in advance the maximum and minimum values to be stored.

Both data types are stored as signed binary integers. INTEGER values have 32 bits and can represent whole numbers from $-2^{31}-1$ through $2^{31}-1$.

SMALLINT values have only 16 bits. They can represent whole numbers from -32,767 through 32,767.

The INT and SMALLINT data types have the following advantages:

- They take up little space (2 bytes per value for SMALLINT and 4 bytes per value for INTEGER).
- You can perform arithmetic expressions such as SUM and MAX and sort comparisons on them.

The disadvantage to using INTEGER and SMALLINT is the limited range of values that they can store. The database server does not store a value that exceeds the capacity of an integer. Of course, such excess is not a problem when you know the maximum and minimum values to be stored.

If you must store a broader range of values that will fill up an INTEGER, you can use BIGINT or INT8. These data types have the following advantages:

- They hold a broad range of values. (Integers ranging from $-(2^{63}-1)$ through $2^{63}-1$.)
- You can perform arithmetic expressions such as SUM and MAX and sort comparisons on them. BIGINT has storage and computational efficiency advantages over INT8.

The disadvantage of using BIGINT or INT8 is that they use more disk space than an INTEGER. The actual size depends on the word length of the platform. An INT8 or SERIAL8 value requires 10 bytes of storage. BIGINT and BIGSERIAL values require 8 bytes of storage.

Automatic sequences: BIGSERIAL, SERIAL, and SERIAL8

The SERIAL data type has the positive non-zero range of an INTEGER with a special feature. Similarly, the BIGSERIAL and SERIAL8 data types have the positive non-zero range of an INT8 with a special feature. Whenever a new row is inserted into a table, the database server automatically generates a new value for BIGSERIAL, SERIAL, or SERIAL8 columns.

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. Because the database server generates the values, the serial values in new rows are always different, even when multiple users are adding rows at the same time. This service is useful because it is quite difficult for an ordinary program to coin unique numeric codes under those conditions. (Informix®, however, also supports sequence objects, which can also support this functionality through the CURRVAL and NEXTVAL operators. For more information about sequence objects, see the description of CREATE SEQUENCE in *Informix® Guide to SQL: Syntax*.

The SERIAL data type can yield up to $2^{31}-1$ positive integers. Consequently, the database server uses all the positive serial numbers by the time it inserts $2^{31}-1$ rows in a table. For most users the exhaustion of the positive serial numbers is not a concern, however, because a single application would have to insert a row every second for 68 years, or 68 applications

would have to insert a row every second for a year, to use all the positive serial numbers. However, if all the positive serial numbers were used, the database server would wrap around and start to generate integer values that begin with a 1.

The BIGSERIAL and SERIAL8 data types can yield up to 2^{63}-1 positive integers. With a reasonable starting value, it is virtually impossible to cause a value of these types to wrap around during insertions.

For these data types, the sequence of generated numbers always increases. When rows are deleted from the table, their serial numbers are not reused. Rows that are sorted on columns of these types are returned in the order in which they were created.

You can specify the initial value in a BIGSERIAL, SERIAL, or SERIAL8 column in the CREATE TABLE statement. This makes it possible to generate different subsequences of system-assigned keys in different tables. The **stores_demo** database uses this technique. In **stores_demo**, the customer numbers begin at 101, and the order numbers start at 1001. If this small business does not register more than 899 customers, all customer numbers have three digits and order numbers have four.

A BIGSERIAL, SERIAL, or SERIAL8 column is not automatically a unique column. If you want to be perfectly sure that no duplicate serial numbers occur, you must apply a unique constraint (see Use CREATE TABLE on page 41). If you define the table using the interactive schema editor in DB-Access, it automatically applies a unique constraint to any BIGSERIAL, SERIAL, or SERIAL8 column.

The BIGSERIAL, SERIAL, and SERIAL8 data types have the following advantages:

- They provide a convenient way to generate system-assigned keys.
- They produce unique numeric codes even when multiple users are updating the table.
- Different tables can use different ranges of numbers.

The BIGSERIAL, SERIAL, and SERIAL8 data types have the following disadvantages:

- A table can have no more than one column of the SERIAL data type, and no more than one column of either the SERIAL8 or BIGSERIAL data type.
- These data types can produce only arbitrary non-NULL positive integer numbers.

For information about the use and behavior of SERIAL, SERIAL8, and BIGSERIAL data types in table hierarchies, see SERIAL types in a table hierarchy on page 126.

Alter the next BIGSERIAL, SERIAL, or SERIAL8 number

The database server sets the starting value for a BIGSERIAL, SERIAL, or SERIAL8 column when it creates the column (see Use CREATE TABLE on page 41). You can use the ALTER TABLE statement later to reset the *next* value, the value that is used for the next inserted row.

You can set the *next* value to any value higher than the current maximum. Doing this will create gaps in the sequence.

If you try to set the *next* value to a value smaller than the highest value currently in the column you will not get an error but the value will not be set. Allowing the *next* value to be set lower than some values in the column would cause duplicate values in some situations and is therefore not allowed.

Approximate numbers: FLOAT and SMALLFLOAT

In scientific, engineering, and statistical applications, numbers are often known to only a few digits of accuracy, and the magnitude of a number is as important as its exact digits.

Floating-point data types are designed for these kinds of applications. They can represent any numeric quantity, fractional or whole, over a wide range of magnitudes from the cosmic to the microscopic. They can easily represent both the average distance from the earth to the sun (1.5×10^{11} meters) or Planck's constant (6.626×10^{-34} joule-seconds). For example,

```
CREATE TABLE t1 (f FLOAT);
INSERT INTO t1 VALUES (0.00000000000000000000000000000000000001);
INSERT INTO t1 VALUES (1.5e11);
INSERT INTO t1 VALUES (6.626196e-34);
```

Two sizes of floating-point data types exist. The FLOAT type is a double-precision, binary floating-point number as implemented in the C language on your computer. A FLOAT data type value usually takes up 8 bytes. The SMALLFLOAT (also known as REAL) data type is a single-precision, binary floating-point number that usually takes up 4 bytes. The main difference between the two data types is their precision.

Floating-point numbers have the following advantages:

- They store very large and very small numbers, including fractional ones.
- They represent numbers compactly in 4 or 8 bytes.
- Arithmetic functions such as AVG, MIN, and sort comparisons are efficient on these data types.

The main disadvantage of floating-point numbers is that digits outside their range of precision are treated as zeros.

Adjustable-precision floating point: DECIMAL(p)

In a database that is not ANSI-compliant, the DECIMAL(p) data type is a floating-point data type similar to FLOAT and SMALLFLOAT. The important difference is that you specify how many significant digits it retains. The precision you write as p can range from 1 to 32, from fewer than SMALLFLOAT up to twice the precision of FLOAT. The magnitude of a DECIMAL(p) number can range from 10^{-130} to 10^{124}. The storage space that DECIMAL(p) numbers use depends on their precision; they occupy $1 + p/2$ bytes (rounded up to a whole number, if necessary).

In an ANSI-compliant database, however, DECIMAL(p) is a fixed-point data type with a scale of zero, so DECIMAL(p) always stores integer values of precision p, if the data value has p or more significant digits. Any fractional part is truncated.

Do not confuse the DECIMAL(p) data type with the DECIMAL(p,s) data type, which is explained in the next section. The DECIMAL(p) data type has only the precision specified.

The DECIMAL(p) data type has the following advantages over FLOAT:

- Precision can be set to suit the application, from approximate to precise.
- Numbers with as many as 32 digits can be represented exactly.
- Storage is used in proportion to the precision of the number.
- Every Informix® database server supports the same precision and range of magnitudes, regardless of the host operating system.

The DECIMAL(*p*) data type has the following disadvantages:

- Performance of arithmetic operations and sorts on DECIMAL(*p*) values is somewhat slower than on FLOAT values.
- Many programming languages do not support the DECIMAL(*p*) data format in the same way that they support FLOAT and INTEGER. When a program extracts a DECIMAL(*p*) value from the database, it might have to convert the value to another format for processing.
- The format and value of a DECIMAL(*p*) data type depends on whether the database is ANSI-compliant.

Fixed-precision numbers: DECIMAL and MONEY

Most commercial applications store numbers that have fixed numbers of digits on the right and left of the decimal point. For example, amounts in U.S. currencies are written with two digits to the right of the decimal point. Normally, you also know the number of digits required on the left, depending on the kinds of transactions that are recorded: perhaps 5 digits for a personal budget, 7 digits for a small business, and 12 or 13 digits for a national budget.

These numbers are *fixed-point* numbers because the decimal point is fixed at a specific place, regardless of the value of the number. The DECIMAL(*p,s*) data type is designed to hold decimal numbers. When you specify a column of this type, you write its *precision* (*p*) as the total number of digits that it can store, from 1 to 32. You write its *scale* (*s*) as the number of those digits that fall to the right of the decimal point. (The following figure shows the relation between precision and scale.) Scale can be zero, meaning it stores only whole numbers. When only whole numbers are stored, DECIMAL(*p,s*) provides a way of storing integers of up to 32 digits.

Figure 14. The relation between precision and scale in a fixed-point number
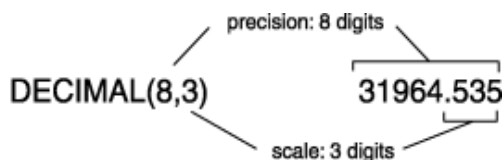
Like the DECIMAL(*p*) data type, DECIMAL(*p,s*) takes up space in proportion to its precision. One value occupies (*p* +3)/2 bytes (if scale is even) or (*p* + 4)/2 bytes (if scale is odd), rounded up to a whole number of bytes.

The MONEY type is identical to DECIMAL(*p,s*) but with one extra feature. Whenever the database server converts a MONEY value to characters for display, it automatically includes a currency symbol.

The advantages of DECIMAL(*p,s*) over INTEGER and FLOAT are that much greater precision is available (up to 32 digits as compared to 10 digits for INTEGER and 16 digits for FLOAT), and both the precision and the amount of storage required can be adjusted to suit the application.

The disadvantages of DECIMAL(*p,s*) are that arithmetic operations are less efficient and that many programming languages do not support numbers in this form. Therefore, when a program extracts a number, it usually must convert the number to another numeric form for processing.

Select a currency format

To customize this currency format, select your locale appropriately or set the **DBMONEY** environment variable. For more information, see the *Informix® Guide to SQL: Reference*.

Global Language Support (GLS)

Each nation has its own way to display money values. When the Informix® database server displays a MONEY value, it refers to a currency format that the user specifies. The default locale specifies a U.S. English currency format of the following form:

`$7,822.45`

For non-English-language locales, you can use the MONETARY category of the locale file to change the current format. For more information about how to use locales, see the *Informix® GLS User's Guide*.

Chronological data types

The chronological data types record time.

The DATE data type stores a calendar date. DATETIME records a point in time to any degree of precision from a year to a fraction of a second. The INTERVAL data type stores a span of time, that is, a duration.

Calendar dates: DATE

The DATE data type stores a calendar date. A DATE value is actually a signed integer whose contents are interpreted as a count of full days since midnight on December 31, 1899.

The DATE format has ample precision to carry dates into the far future (58,000 centuries). Negative DATE values are interpreted as counts of days before the epoch date; that is, a DATE value of -1 represents December 30, 1899.

Because DATE values are integers, the values can be used in arithmetic expressions. For example, you can take the average of a DATE column, or you can add 7 or 365 to a DATE column. In addition, a rich set of functions exists specifically for manipulating DATE values. For more information, see the *Informix® Guide to SQL: Syntax*.

The DATE data type is compact, at 4 bytes per item. Arithmetic functions and comparisons execute quickly on a DATE column.

Select a date format (GLS)

You can punctuate and order the components of a date in many ways. When an application displays a DATE value, it refers to a date format that the user specifies. The default locale specifies a U.S. English date format of the form: `10/25/2001`

To customize this date format, select your locale appropriately or set the **DBDATE** environment variable. For more information, see the *Informix® Guide to SQL: Reference*.

For non-default locales, you can use the **GL_DATE** environment variable to specify the date format. For more information about how to use locales, see the *Informix® GLS User's Guide*.

Exact points in time: DATETIME

The DATETIME data type stores any moment in time in the era that began 1 A.D. In fact, DATETIME is really a family of 28 data types, each with a different precision. When you define a DATETIME column, you specify its precision. The column can contain any sequence from the list:

- year
- month
- day
- hour
- minute
- second
- *fraction*

Thus, you can define a DATETIME column that stores only a year, only a month and day, or a date and time that is exact to the hour or even to the millisecond. The following table shows that the size of a DATETIME value ranges from 2 to 11 bytes depending on its precision.

The advantage of DATETIME is that it can store specific date and time values. A DATETIME column typically requires more storage space than a DATE column, depending on the DATETIME qualifiers. Datetime also has an inflexible display format. For information about how to circumvent the display format, see .

Table 1. Precisions for the DATETIME data type

Precision	Size (When f is odd, round the size to the next full byte)	Precision	Size (When f is odd, round the size to the next full byte)
year to year	3	day to hour	3
year to month	4	day to minute	4
year to day	5	day to second	5
year to hour	6	day to fraction(f)	$5 + f/2$
year to minute	7	hour to hour	2
year to second	8	hour to minute	3
year to fraction (f)	$8 + f/2$	hour to second	4

Table 1. Precisions for the DATETIME data type (continued)

Precision	Size (When f is odd, round the size to the next full byte)	Precision	Size (When f is odd, round the size to the next full byte)
month to month	2	hour to fraction(f)	$4 + f/2$
month to day	3	minute to minute	2
month to hour	4	minute to second	3
month to minute	5	minute to fraction(f)	$3 + f/2$
month to second	6	second to second	2
month to fraction(f)	$6 + f/2$	second to fraction(f)	$2 + f/2$
day to day	2	fraction to fraction(f)	$1 + f/2$

Durations using INTERVAL

The INTERVAL data type stores a duration, that is, a length of time. The difference between two DATETIME values is an INTERVAL, which represents the span of time that separates them. The following examples might help to clarify the differences:

- An employee began working on January 21, 1997 (either a DATE or a DATETIME).
- They have worked for 254 days (an INTERVAL value, the difference between the TODAY function and the starting DATE or DATETIME value).
- They begin work each day at 0900 hours (a DATETIME value).
- They work 8 hours (an INTERVAL value) with 45 minutes for lunch (another INTERVAL value).
- Their quitting time is 1745 hours (the sum of the DATETIME when they begin work and the two INTERVALs).

Like DATETIME, INTERVAL is a family of data types with different precisions. An INTERVAL value can represent a count of years and months, or it can represent a count of days, hours, minutes, seconds, or fractions of seconds; 18 precisions are possible. The size of an INTERVAL value ranges from 2 to 12 bytes, depending on the formulas that shows.

Table 2. Precisions for the INTERVAL data type.

Precision	Size (Round a fractional size to the next full byte)	Precision	Size (Round a fractional size to the next full byte)
year(p) to year	$1 + p/2$	hour(p) to minute	$2 + p/2$

Table 2. Precisions for the INTERVAL data type. (continued)

Precision	Size (Round a fractional size to the next full byte)	Precision	Size (Round a fractional size to the next full byte)
year(p) to month	2 + p/2	hour(p) to second	3 + p/2
month(p) to month	1 + p/2	hour(p) to fraction(f)	4 + (p + f)/2
day(p) to day	1 + p/2	minute(p) to minute	1 + p/2
day(p) to hour	2 + p/2	minute(p) to second	2 + p/2
day(p) to minute	3 + p/2	minute(p) to fraction(f)	3 + (p + f)/2
day(p) to second	4 + p/2	second(p) to second	1 + p/2
day(p) to fraction(f)	5 + (p + f)/2	second(p) to fraction(f)	2 + (p + f)/2
hour(p) to hour	1 + p/2	fraction to fraction(f)	1 + f/2

INTERVAL values can be negative or positive. You can add or subtract them, and you can scale them by multiplying or dividing by a number. This is not true of either DATE or DATETIME. You can reasonably ask, What is one-half the number of days until April 23? but not, What is one-half of April 23?

Force the format of a DATETIME or INTERVAL value

The database server always displays the components of an INTERVAL or DATETIME value in the order `year-month-day hour:minute:second.fraction`. It does not refer to the date format that is defined to the operating system, as it does when it formats a DATE value.

You can write a SELECT statement that displays the date part of a DATETIME value in the system-defined format. The trick is to isolate the component fields with the **EXTEND** function and pass them through the **MDY()** function, which converts them to a DATE. The following code shows a partial example:

```
SELECT ... MDY (
   EXTEND (DATE_RECEIVED, MONTH TO MONTH),
   EXTEND (DATE_RECEIVED, DAY TO DAY),
   EXTEND (DATE_RECEIVED, YEAR TO YEAR) )
   FROM RECEIPTS ...
```

Select a DATETIME format (GLS)

When an application displays a DATETIME value, it refers to a DATETIME format that the user specifies. The default locale specifies a U.S. English DATETIME format of the following form: `2001-10-25 18:02:13`

For non-default locales, you can use the **GL_DATETIME** environment variable to specify the DATETIME format. For more information about how to use locales, see the *Informix® GLS User's Guide*.

To customize this DATETIME format, select your locale appropriately or set the **GL_DATETIME** or **DBTIME** environment variable. For more information about these environment variables, see the *Informix® GLS User's Guide*.

BOOLEAN data type

The BOOLEAN data type is a 1-byte data type. The legal values for Boolean are true ('t'), false ('f'), or NULL. The values are not case sensitive.

You can compare a BOOLEAN column against another BOOLEAN column or against Boolean values. For example, you might use these SELECT statements:

```
SELECT * FROM sometable WHERE bool_col = 't';
SELECT * FROM sometable WHERE bool_col IS NULL;
```

Character data types (GLS)

Informix® database servers support several character data types, including CHAR, NCHAR, and NVARCHAR, the special-use character data type.

Character data: CHAR(n) and NCHAR(n)

The CHAR(n) data type contains a sequence of n bytes. These characters can be a mixture of English-language and non-English-language characters and can be either single byte or multibyte (Asian). The length n ranges from 1 to 32,767.

Whenever the database server retrieves or stores a CHAR(n) value, it transfers exactly n bytes. If an inserted value is shorter than n, the database server extends the value with single-byte ASCII space characters to make up n bytes. If an inserted value exceeds n bytes, the database server truncates the extra characters without returning an error message. Thus the semantic integrity of data for a CHAR(n) column or variable is not enforced when the value that is inserted or updated exceeds n bytes.

Data in CHAR columns is sorted in code-set order. For example, in the ASCII code set, the character *a* has a code-set value of 97, *b* has 98, and so forth. The database server sorts CHAR(n) data in this order.

The NCHAR(n) data type also contains a sequence of n bytes. These characters can be a mixture of English-language and non-English-language characters and can be either single byte or multibyte (Asian). The length of n has the same limits as the CHAR(n) data type. Whenever an NCHAR(n) value is retrieved or stored, exactly n bytes are transferred. The number of characters transferred can be fewer than the number of bytes if the data contains multibyte characters. If an inserted value is shorter than n, the database server extends the value with space characters to make up n bytes.

The database server sorts data in NCHAR(n) columns according to the order that the locale specifies. For more information about how to use locales, refer to the *Informix® GLS User's Guide*.

Tip: The only difference between CHAR(*n*) and NCHAR(*n*) data is how you sort and compare the data. You can store non-English-language characters in a CHAR(*n*) column. However, because the database server uses code-set order to perform any sorting or comparison on CHAR(*n*) columns, you might not obtain the results in the order that you expect.

A CHAR(*n*) or NCHAR(*n*) value can include tabs and spaces but normally contains no other nonprinting characters. When you insert rows with INSERT or UPDATE, or when you load rows with a utility program, no means exist for entering nonprintable characters. However, when a program that uses embedded SQL creates rows, the program can insert any character except the null (binary zero) character. It is not a good idea to store nonprintable characters in a character column because standard programs and utilities do not expect them.

The advantage of the CHAR(*n*) or NCHAR(*n*) data type is its availability on all database servers. The only disadvantage of CHAR(*n*) or NCHAR(*n*) is its fixed length. When the length of data values varies widely from row to row, space is wasted.

Variable-length strings: CHARACTER VARYING(*m,r*), VARCHAR(*m,r*), NVARCHAR(*m,r*), and LVARCHAR(*m*)

Often the data values in a character column are different lengths. That is, many are an average length, and only a few are the maximum length. For the following data types, *m* represents the maximum number of bytes and *r* represents a minimum number of bytes that the column stores. These *variable-length data types* are designed to save disk space when you store such data:

CHARACTER VARYING (*m,r*)

> The CHARACTER VARYING (*m,r*) data type contains a sequence of, at most, *m* bytes or at the least, *r* bytes. This data type is the ANSI-compliant format for character data of varying length. CHARACTER VARYING (*m,r*), supports code-set order for comparisons of its character data.

VARCHAR (*m,r*)

> The VARCHAR (*m,r*) is one of the Informix®-specific data types for storing character data of varying length. In functionality, it is the same as CHARACTER VARYING(*m,r*).

NVARCHAR (*m,r*)

> The NVARCHAR (*m,r*) is one of theInformix®-specific data types for storing character data of varying length. It compares character data in the order that the locale specifies.

LVARCHAR(*m*)

> The LVARCHAR is the Informix®-specific data type for storing character data of varying length from 1 to 32,739 bytes. If no maximum size is specified in the declaration of a column length, the default is 2,048 bytes. LVARCHAR supports code-set order for collation, and is also used by the database server for internal operations on character strings, whose maximum size is operating-system dependent.

Tip: The difference in the way data is compared distinguishes NVARCHAR(*m,r*) data from CHARACTER VARYING(*m,r*) or VARCHAR(*m,r*) data. For more information about how the locale determines code-set and sort order, see Character data: CHAR(n) and NCHAR(n) on page 33.

In databases that are created as NLSCASE INSENSITIVE, only the NCHAR and NVARCHAR data types are processed by the database server without regard to letter-case variants, so that (for example) the NCHAR strings 'pH' and 'Ph' are treated as duplicate values in ordering, sorting, and comparison operations.

When you define columns as variable-length data types, you can specify *m* as the *maximum* number of bytes. If an inserted value consists of fewer than *m* bytes, the database server does not extend the value with single-byte spaces (as with CHAR(*n*) and NCHAR(*n*) values). Instead, it stores only the actual contents on disk with a 1-byte length field. The limit on *m* is 254 bytes for indexed columns and 255 bytes for non-indexed columns.

The second parameter, *r*, is an optional *reserve* length that sets a lower limit on the number of bytes than a value being stored on disk requires. Even if a value requires fewer than *r* bytes, *r* bytes are nevertheless allocated to hold it. The purpose is to save time when rows are updated. (See Variable-length execution time on page 35.) The LVARCHAR data type supports no *reserve* length.

The advantages of the CHARACTER VARYING(*m,r*), LVARCHAR(*m*), or VARCHAR(*m,r*) data type over the CHAR(*n*) data type are as follows:

- They conserve disk space when the number of bytes that data items require varies widely, or when only a few items require more bytes than average.
- Queries on the more compact tables can be faster.

These advantages also apply to the NVARCHAR(*m,r*) data type in comparison to the NCHAR(*n*) data type.

The following are potential disadvantages of using varying-length data types:

- Except for LVARCHAR, they do not support lengths that exceed 255 bytes.
- Table updates can be slower in some circumstances.

Variable-length execution time

When you use any of the CHARACTER VARYING(*m,r*), VARCHAR(*m,r*), or NVARCHAR(*m,r*) data types, the rows of a table have a varying number of bytes instead of a fixed number of bytes. The speed of database operations is affected when the rows of a table have varying numbers of bytes.

Because more rows fit in a disk page, the database server can search the table with fewer disk operations than if the rows were of a fixed number of bytes. As a result, queries can execute more quickly. Insert and delete operations can be a little quicker for the same reason.

When you update a row, the amount of work the database server must perform depends on the number of bytes in the new row as compared with the number of bytes in the old row. If the new row uses the same number of bytes or fewer, the execution time is not significantly different than it is with fixed-length rows. However, if the new row requires a greater number of bytes than the old one, the database server might have to perform several times as many disk operations. Thus,

updates of a table that use CHARACTER VARYING(*m,r*), VARCHAR(*m,r*), or NVARCHAR(*m,r*) data can sometimes be slower than updates of a fixed-length field.

To mitigate this effect, specify *r* as a number of bytes that encompasses a high proportion of the data items. Then most rows use the reserve number of bytes, and padding wastes only a little space. Updates are slow only when a value that uses the reserve number of bytes is replaced with a value that uses more than the reserve number of bytes.

Large character objects: TEXT

The TEXT data type stores a block of text. It is designed to store self-contained documents: business forms, program source or data files, or memos. Although you can store any data in a TEXT item, Informix® tools expect a TEXT item to be printable, so restrict this data type to printable ASCII text.

TEXT values are not stored with the rows of which they are a part. They are allocated in whole disk pages, usually in areas separate from rows. For more information, see your *Informix® Administrator's Guide*.

The advantage of the TEXT data type over CHAR(*n*) and VARCHAR(*m,r*) is that the size of a TEXT data item has no limit except the capacity of disk storage to hold it. The disadvantages of the TEXT data type are as follows:

- It is allocated in whole disk pages, so a short item wastes space.
- Restrictions apply on how you can use a TEXT column in an SQL statement. (For more information about this restriction, see Use TEXT and BYTE data types on page 36.)

Binary objects: BYTE

The BYTE data type is designed to hold any data a program can generate: graphic images, program object files, and documents saved by any word processor or spreadsheet. The database server permits any kind of data of any length in a BYTE column.

As with TEXT, BYTE data items usually are stored in whole disk pages in disk areas separate from normal row data.

The advantage of the BYTE data type, as opposed to TEXT or CHAR(*n*), is that it accepts any data. Its disadvantages are the same as those of the TEXT data type.

Use TEXT and BYTE data types

The database server stores and retrieves TEXT and BYTE columns. To fetch and store TEXT or BYTE values, you normally use programs written in a language that supports embedded SQL, such as Informix® ESQL/C. In such a program, you can fetch, insert, or update a TEXT or BYTE value in a manner similar to the way you read or write a sequential file.

In no SQL statement, interactive or programmed, can a TEXT or BYTE column be used in the following ways:

- In arithmetic or Boolean expressions
- In a GROUP BY or ORDER BY clause

- In a UNIQUE test
- For indexing, either by itself or as part of a composite index

In a SELECT statement that you enter interactively or in a form or report, you can perform the following operations on a TEXT or BYTE value:

- Select the column name, optionally with a subscript to extract part of it.
- Use LENGTH(*column_name*) to return the length of the column.
- Test the column with the IS [NOT] NULL predicate.

In an interactive INSERT statement, you can use the VALUES clause to insert a TEXT or BYTE value, but the only value that you can give that column is null. However, you can use the SELECT form of the INSERT statement to copy a TEXT or BYTE value from another table.

In an interactive UPDATE statement, you can update a TEXT or BYTE column to null or to a subquery that returns a TEXT or BYTE column.

Change the data type

After the table is built, you can use the ALTER TABLE statement to change the data type that is assigned to a column. Although such alterations are sometimes necessary, you should avoid them for the following reasons:

- To change a data type, the database server must copy and rebuild the table. For large tables, copying and rebuilding can take a lot of time and disk space.
- Some data type changes can cause a loss of information. For example, when you change a column from a longer to a shorter character type, long values are truncated; when you change to a less-precise numeric type, low-order digits are truncated.
- Existing programs, forms, reports, and stored queries might also have to be changed.

Null values

In most cases, columns in a table can contain null values. A null value means that the value for the column can be unknown or not applicable. For example, in the telephone directory example in Build a relational data model on page 6, the **anniv** column of the **name** table can contain null values; if you do not know the person's anniversary, you do not specify it. Do not confuse null value with zero or a blank value. For example, the following statement inserts a row into the **manufact** table of the **stores_demo** database and specifies that the value for the **lead_time** column is null:

```
INSERT INTO manufact VALUES ('DRM', 'Drumm', NULL)
```

Collection columns cannot contain null elements. Create and use extended data types in Informix on page 97 describes collection data types.

Default values

A *default value* is the value that is inserted into a column when an explicit value is not specified in an INSERT statement. A default value can be a literal character string that you define or one of the following SQL constant expressions:

- USER
- CURRENT
- TODAY
- DBSERVERNAME

Not all columns require default values, but as you work with your data model, you might discover instances where the use of a default value saves data-entry time or prevents data-entry error. For example, the telephone directory model has a **state** column. While you look at the data for this column, you discover that more than 50 percent of the addresses list California as the state. To save time, specify the string `CA` as the default value for the **state** column.

Check constraints

Check constraints specify a condition or requirement on a data value before data can be assigned to a column during an INSERT or UPDATE statement. If a row evaluates to *false* for any of the check constraints that are defined on a table during an insert or update, the database server returns an error. However, the database server does not report an error or reject the record when the check constraint evaluates to NULL. For this reason, you might want to use both a check constraint and a NOT NULL constraint when you create a table.

To define a constraint, use the CREATE TABLE or ALTER TABLE statements. For example, the following requirement constrains the values of an integer domain to a certain range:

```
Customer_Number >= 50000 AND Customer_Number <= 99999
```

To express constraints on character-based domains, use the MATCHES predicate and the regular-expression syntax that it supports. For example, the following constraint restricts a telephone domain to the form of a U.S. local telephone number:

```
vce_num MATCHES '[2-9][2-9][0-9]-[0-9][0-9][0-9][0-9]'
```

For additional information about check constraints, see the CREATE TABLE and ALTER TABLE statements in the *Informix® Guide to SQL: Syntax*.

Referential constraints

You can identify the primary and foreign keys in each table to place referential constraints on columns. Build a relational data model on page 6 contains information about how you identify these keys.

When you are trying to pick columns for primary and foreign keys, almost all data type combinations must match. For example, if you define a primary key as a CHAR data type, you must also define the foreign key as a CHAR data type.

However, when you specify a SERIAL data type on a primary key in one table, you specify an INTEGER on the foreign key of the relationship. Similarly, when you specify a SERIAL8 data type on a primary key in one table, you specify an INT8 on the foreign key of the relationship; and when you specify a BIGSERIAL data type on a primary key in one table, you specify a BIGINT data type on the foreign key of the relationship.

Data type combinations that you can mix in a relationship are as follows (1st column is the foreign key and 2nd column is the primary key):

- INTEGER and SERIAL
- INTEGER and BIGINT
- INTEGER and BIGSERIAL
- INTEGER and INT8
- INTEGER and SERIAL8
- SERIAL and INTEGER
- SERIAL and BIGINT
- SERIAL and BIGSERIAL
- SERIAL and INT8
- SERIAL and SERIAL8
- INT8 and BIGINT
- INT8 and SERIAL8
- INT8 and BIGSERIAL
- SERIAL8 and INT8
- SERIAL8 and BIGINT
- SERIAL8 and BIGSERIAL
- BIGINT and INT8
- BIGINT and BIGSERIAL
- BIGINT and SERIAL8
- BIGSERIAL and BIGINT
- BIGSERIAL and INT8
- BIGSERIAL and SERIAL8

For information about how to create a table with referential constraints, see the CREATE TABLE and ALTER TABLE statements in the *Informix® Guide to SQL: Syntax*.

Implement a relational data model

This chapter shows how to use SQL syntax to implement the data model that Build a relational data model on page 6 describes. In other words, it shows you how to create a database and tables and populate the tables with data. This chapter also contains information about database logging options, table synonyms, and command scripts.

Create the database

Now you are ready to create the data model as tables in a database. You do this with the CREATE DATABASE, CREATE TABLE, and CREATE INDEX statements. The syntax for these statements is described in the *Informix® Guide to SQL: Syntax*. This section contains information about how to use the CREATE DATABASE and CREATE TABLE statements to implement a data model.

Remember that the telephone directory data model is used for illustrative purposes only. For the sake of the example, it is translated into SQL statements.

You might have to create the same database model more than once. You can store the statements that create the model and later re-execute those statements. For more information, see Use command scripts on page 46.

When the tables exist, you must populate them with rows of data. You can do this manually, with a utility program, or with custom programming.

Use CREATE DATABASE

A database is a container that holds all parts of a data model. These parts include not only the tables but also views, indexes, synonyms, and other objects that are associated with the database. You must create a database before you can create anything else.

When the database server creates a database, it stores the locale of the database that is derived from the **DB_LOCALE** environment variable in its system catalog. This locale determines how the database server interprets character data that is stored within the database. By default, the database locale is the U.S. English locale that uses the ISO8859-1 code set. For information about how to use alternative locales, see the *Informix® GLS User's Guide*.

When the database server creates a database, it sets up records that show the existence of the database and its mode of logging. These records are not visible to operating-system commands because the database server manages disk space directly.

Avoid name conflicts

Normally, only one copy of the database server is running on a computer, and the database server manages the databases that belong to all users of that computer. The database server keeps only one list of database names. The name of your database must be different from that of any other database that the database server manages. (It is possible to run more than one copy of the database server. You can create more than one copy of the database server, for example, to create a safe environment for testing apart from the operational data. In this case, be sure that you are using the correct database server when you create the database and again when you access it later.)

Select a dbspace

The database server lets you create the database in a particular *dbspace*. A dbspace is a named area of disk storage. Ask your database server administrator whether you should use a particular dbspace. You can put a database in a separate dbspace to isolate it from other databases or to locate it on a particular disk device. For information about dbspaces and their relationship to disk devices, see your *Informix® Administrator's Guide*.

Some dbspaces are *mirrored* (duplicated on two disk devices for high reliability). You might put your database in a mirrored dbspace if its contents are of exceptional importance.

Select the type of logging

To specify a logging or nonlogging database, use the CREATE DATABASE statement. The database server offers the following choices for transaction logging:

- No logging at all.

  This is not a recommended choice. If you lose the database because of a hardware failure, you lose all data alterations since the last backup.

  ```
  CREATE DATABASE db_with_no_log
  ```

  When you do not select logging, BEGIN WORK and other SQL statements that are related to transaction processing are not permitted in the database. This situation affects the logic of programs that use the database.
- Regular (unbuffered) logging.

  This choice is best for most databases. In the event of a failure, you lose only uncommitted transactions.

  ```
  CREATE DATABASE a_logged_db WITH LOG
  ```
- Buffered logging.

  If you lose the database, you lose few or possibly none of the most recent alterations. In return for this small risk, performance during alterations improves slightly.

  ```
  CREATE DATABASE buf_log_db WITH BUFFERED LOG
  ```

  Buffered logging is best for databases that are updated frequently (so that speed of updating is important), but you can re-create the updates from other data in the event of a failure. Use the SET LOG statement to alternate between buffered and regular logging.
- ANSI-compliant logging.

  This logging is the same as regular logging, but the ANSI rules for transaction processing are also enforced. For more information, see Use ANSI-compliant databases on page 2.

  ```
  CREATE DATABASE std_rules_db WITH LOG MODE ANSI
  ```

  The design of ANSI SQL prohibits the use of buffered logging. When you create an ANSI-compliant database, you cannot turn off transaction logging.

For databases that are not ANSI-compliant, the database server administrator (DBA) can turn transaction logging on and off or change from buffered to unbuffered logging. For example, you might turn logging off before inserting a large number of new rows.

You can use the ondblog and ontape utilities to change the logging status or buffering mode. For information about these tools, see the *Informix® Administrator's Guide*. You can also use the SET LOG statement to change between buffered and unbuffered logging. For information about SET LOG, see your *Informix® Guide to SQL: Syntax*.

Use CREATE TABLE

Use the CREATE TABLE statement to create each table that you design in the data model. This statement has a complicated form, but it is basically a list of the columns of the table. For each column, you supply the following information:

- The name of the column
- The data type (from the domain list you made)

The statement might also contain one or more of the following constraints:

- A primary-key constraint
- A foreign-key constraint
- A NOT NULL constraint (or a NULL constraint, allowing NULL values)
- A unique constraint
- A default constraint
- A check constraint

In short, the CREATE TABLE statement is an image, in words, of the table as you drew it in the data-model diagram in . The following example shows the statements for the telephone directory data model:

```
CREATE TABLE name
   (
   rec_num  SERIAL PRIMARY KEY,
   lname    CHAR(20),
   fname    CHAR(20),
   bdate    DATE,
   anniv    DATE,
   email    VARCHAR(25)
   );

CREATE TABLE child
   (
   child    CHAR(20),
   rec_num  INT,
   FOREIGN KEY (rec_num) REFERENCES NAME (rec_num)
   );

CREATE TABLE address
   (
   id_num    SERIAL PRIMARY KEY,
   rec_num   INT,
   street    VARCHAR (50,20),
   city      VARCHAR (40,10),
   state     CHAR(5) DEFAULT 'CA',
   zipcode   CHAR(10),
   FOREIGN KEY (rec_num) REFERENCES name (rec_num)
   );

CREATE TABLE voice
   (
   vce_num    CHAR(13) PRIMARY KEY,
   vce_type   CHAR(10),
   rec_num    INT,
   FOREIGN KEY (rec_num) REFERENCES name (rec_num)
   );

CREATE TABLE fax
```

```
    (
    fax_num    CHAR(13),
    oper_from  DATETIME HOUR TO MINUTE,
    oper_till  DATETIME HOUR TO MINUTE,
    PRIMARY KEY (fax_num)
    );

CREATE TABLE faxname
    (
    fax_num    CHAR(13),
    rec_num    INT,
    PRIMARY KEY (fax_num, rec_num),
    FOREIGN KEY (fax_num) REFERENCES fax (fax_num),
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
    );

CREATE TABLE modem
    (
    mdm_num    CHAR(13) PRIMARY KEY,
    rec_num    INT,
    b_type     CHAR(5),
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
    );
```

In each of the preceding examples, the table data gets stored in the same dbspace that you specify for the database because the CREATE TABLE statement does not specify a storage option. You can specify a dbspace for the table that is different from the storage location of the database or fragment the table into multiple dbspaces. For information about the different storage options Informix® database servers support, see the CREATE TABLE statement in the *Informix® Guide to SQL: Syntax*. The following section shows one way to fragment a table into multiple dbspaces.

Create a fragmented table

To control where data is stored at the table level, you can use a FRAGMENT BY clause when you create the table. The following statement creates a fragmented table that stores data according to a round-robin distribution scheme. In this example, the rows of data are distributed more or less evenly across the fragments `dbspace1`, `dbspace2`, and `dbspace3`.

```
CREATE TABLE name
    (
    rec_num  SERIAL PRIMARY KEY,
    lname    CHAR(20),
    fname    CHAR(20),
    bdate    DATE,
    anniv    DATE,
    email    VARCHAR(25)
    ) FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

For more information about the different distribution schemes that you can use to create fragmented tables, see Table fragmentation strategies on page 50.

Drop or modify a table

Use the DROP TABLE statement to remove a table with its associated indexes and data. To change the definition of a table, for example, by adding a check constraint, use the ALTER TABLE statement. Use the TRUNCATE statement to remove all rows from a table and all corresponding index data while preserving the definition of the table. For information about these statements, see *Informix® Guide to SQL: Syntax*.

Use CREATE INDEX

Use the CREATE INDEX statement to create an index on one or more columns in a table and, optionally, to cluster the physical table in the order of the index. This section describes some of the options available when you create indexes. For more information about the CREATE INDEX statement, see the *Informix® Guide to SQL: Syntax*.

Suppose you create table **customer**:

```
CREATE TABLE customer
(
   cust_num    SERIAL(101) UNIQUE
   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)
);
```

The following statement shows how to create an index on the **lname** column of the **customer** table:

```
CREATE INDEX lname_index ON customer (lname);
```

Composite indexes

You can create an index that includes multiple columns. For example, you might create the following index:

```
CREATE INDEX c_temp2 ON customer (cust_num, zipcode);
```

Bidirectional traversal of indexes

The ASC and DESC keywords specify the order in which the database server maintains the index. When you create an index on a column and omit the keywords or specify the ASC keyword, the database server stores the key values in ascending order. If you specify the DESC keyword, the database server stores the key values in descending order.

Ascending order means that the key values are stored in order from the smallest key to the largest key. For example, if you create an ascending index on the **lname** column of the **customer** table, last names are stored in the index in the following order: `Albertson, Beatty, Currie`.

Descending order means that the key values are stored in order from the largest key to the smallest key. For example, if you create a descending index on the **lname** column of the **customer** table, last names are stored in the index in the following order: `Currie, Beatty, Albertson`.

The bidirectional traversal capability of the database server lets you create just one index on a column and use that index for queries that specify sorting of results in either ascending or descending order of the sort column.

Use synonyms for table names

A *synonym* is a name that you can use in place of another SQL identifier. You use the CREATE SYNONYM statement to declare an alternative name for a table, a view, or (for) a sequence object.

Typically, you use a synonym to refer to tables that are not in the current database. For example, you might execute the following statements to create synonyms for the **customer** and **orders** table names:

```
CREATE SYNONYM mcust FOR masterdb@central:customer;
CREATE SYNONYM bords FOR sales@boston:orders;
```

After you create the synonym, you can use it in many contexts where the original table name is valid, as the following example shows:

```
SELECT bords.order_num, mcust.fname, mcust.lname
   FROM mcust, bords
   WHERE mcust.customer_num = bords.Customer_num
   INTO TEMP mycopy;
```

The CREATE SYNONYM statement stores the synonym name in the system catalog table **syssyntable** in the current database. The synonym is available to any query made in that database. (If the **USETABLENAME** environment variable is set, however, some DDL statements of SQL do not support synonyms in place of table names.)

A short synonym makes it easier to write queries, but synonyms can play another role. They allow you to move a table to a different database, or even to a different computer, and keep your queries the same.

Suppose you have several queries that refer to the tables **customer** and **orders**. The queries are embedded in programs, forms, and reports. The tables are part of the demonstration database, which is kept on database server **avignon**.

Now you decide to make the same programs, forms, and reports available to users of a different computer on the network (database server **nantes**). Those users have a database that contains a table named **orders** that contains the orders at their location, but they must have access to the table **customer** at **avignon**.

To those users, the **customer** table is external. Does this mean you must prepare special versions of the programs and reports, versions in which the **customer** table is qualified with a database server name? A better solution is to create a synonym in the users' database, as the following example shows:

```
DATABASE stores_demo@nantes;
CREATE SYNONYM customer FOR stores_demo@avignon:customer;
```

When the stored queries are executed in your database, the name **customer** refers to the actual table. When they are executed in the other database, the name is resolved through the synonym into a reference to the table that exists on the database server **avignon**. (In a database that is not ANSI-compliant, a synonym must be unique among the names

of synonyms, tables, views, and sequence objects in the database. In an ANSI-compliant database, the *owner.synonym* combination must be unique within the namespace of objects that have been registered in the database with a **tabid** value.)

Use synonym chains

To continue the preceding example, suppose that a new computer is added to your network. Its name is **db_crunch**. The **customer** table and other tables are moved to it to reduce the load on **avignon**. You can reproduce the table on the new database server easily enough, but how can you redirect all access to it? One way is to install a synonym to replace the old table, as the following example shows:

```
DATABASE stores_demo@avignon EXCLUSIVE;
RENAME TABLE customer TO old_cust;
CREATE SYNONYM customer FOR stores_demo@db_crunch:customer;
CLOSE DATABASE;
```

When you execute a query within **stores_demo@avignon**, a reference to table **customer** finds the synonym and is redirected to the version on the new computer. Such redirection also happens for queries that are executed from database server **nantes** in the previous example. The synonym in the database **stores_demo@nantes** still redirects references to **customer** to database **stores_demo@avignon**; the new synonym there sends the query to database **stores_demo@db_crunch**.

Chains of synonyms can be useful when, as in this example, you want to redirect all access to a table in one operation. However, you should update the databases of all users as soon as possible so their synonyms point directly to the table. If you do not, you incur extra overhead when the database server handles the extra synonyms, and the table cannot be found if any computer in the chain is down.

You can run an application against a local database and later run the same application against a database on another computer. The program runs equally well in either case (although it can run more slowly on the network database). If the data model is the same, a program cannot tell the difference between one database and another.

Use command scripts

You can enter SQL statements interactively to create the database and tables. In some cases, you might have to create the database and tables two or more times. For example, you might have to create the database again to make a production version after a test version is satisfactory, or you might have to implement the same data model on several computers. To save time and reduce the chance of errors, you can put all the statements to create a database in a file and later re-execute those statements.

Capture the schema

The **dbschema** utility is a program that examines the contents of a database and generates all the SQL statements you require to re-create it. You can build the first version of your database, making changes until it is exactly as you want it. Then

you can use **dbschema** to generate the SQL statements necessary to duplicate it. For information about the **dbschema** utility, see the *Informix® Migration Guide*.

Execute the file

Programs that you use to enter SQL statements interactively, such as DB-Access, can be run from a file of commands. You can start DB-Access to read and execute a file of commands that you or **dbschema** prepared. For more information, see the *Informix® DB-Access User's Guide*.

An example

Most Informix® database server products come with a demonstration database (the database that most of the examples in this book use). The demonstration database is delivered as an operating-system command script that calls Informix® products to build the database. You can copy this command script and use it as the basis to automate your own data model.

Populate the database

For your initial tests, the easiest way to populate the database is to type INSERT statements in DB-Access. For example, to insert a row into the **manufact** table of the demonstration database, enter the following command in DB-Access:

```
INSERT INTO manufact VALUES ('MKL', 'Martin', 15);
```

If you are preparing an application program, such as an application in C, you can use the application to enter rows into a database table.

The following table lists Informix® tools that you can use for entering information into your database. The acronyms in the Reference column are explained after the table.

Tool	Purpose	Reference
dbaccessdemo	Prepare and populate sample databases.	DB-A, SQLR
DB-Access	Edit a database by entering explicit commands.	DB-A, SQLS
onunload & onload	Copy an entire database or selected database tables to or from files on tape or disk.	MG, AR
dbload	Load data from one or more text files into one or more existing tables.	MG
LOAD & UNLOAD	Load data from (or into) a text file.	SQLS
dbexport , dbimport	Copy an entire database using text files.	MG
Enterprise Replication	Update selected databases each time a specified table is updated.	ER

Tool	Purpose	Reference
C application	Use SQL commands embedded in a C program to update databases.	ESQLC, DAPI
Java™ application	Use SQL commands embedded in a Java™ program to update databases.	Java™

Mnemonic

    Explanation of References Column

SQLR

    Informix® Guide to SQL: Reference

SQLS

    Informix® Guide to SQL: Syntax

MG

    Informix® Migration Guide

AR

ESQL/C

    Informix® ESQL/C Programmer's Manual

Java™

    J/Foundation Developer's Guide

DB-A

    Informix® DB-Access User's Guide

ER

DAPI

    Informix® DataBlade® API Programmer's Guide

Move data from other HCL Informix® databases

Often, the initial rows of a table can be derived from data that is stored in tables in another Informix® database or in operating-system files. The following utilities let you move large quantities of data:

- onunload and onload utilities
- dbexport and dbimport utilities
- dbload utility
- SQL LOAD statement

You can also select the data you want from the other database on another database server as part of an INSERT statement in your database. As the following example shows, you can select information from the **items** table in the demonstration database to insert into a new table:

```
INSERT INTO newtable
   SELECT item_num, order_num, quantity, stock_num,
     manu_code, total_price
   FROM stores_demo@otherserver:items;
```

Load source data into a table

When the data source is not the Informix® database, you must find a way to convert it into a flat ASCII file; that is, a file of printable data in which each line represents the contents of one table row.

After you have the data in an ASCII file, you can use the dbload utility to load it into a table. For more information about dbload, see the *Informix® Migration Guide*. The LOAD statement in DB-Access can also load rows from a flat ASCII file. For information about the LOAD and UNLOAD statements, see the *Informix® Guide to SQL: Syntax*.

After you have the data in a file, you can use *external tables* to load it into a table. For more information about external tables, see your *Informix® Administrator's Guide*.

Perform bulk-load operations

Inserting hundreds or thousands of rows goes much faster if you turn off transaction logging. Logging these insertions makes no sense because, in the event of a failure, you can easily re-create the lost work. The following list contains the steps of a large bulk-load operation:

- If any chance exists that other users are using the database, exclude them with the DATABASE EXCLUSIVE statement.
- Ask the administrator to turn off logging for the database.

  The existing logs can be used to recover the database in its present state, and you can run the bulk insertion again to recover those rows if they are lost.
- Perform the statements or run the utilities that load the tables with data.
- Back up the newly loaded database.

  Either ask the administrator to perform a full or incremental backup or use the **onunload** utility to make a binary copy of your database only.
- Restore transaction logging and release the exclusive lock on the database.

Managing databases

Table fragmentation strategies

This chapter describes the fragmentation strategies that your database server supports and provides examples of the different fragmentation strategies. It contains information about fragmentation, distribution schemes for table fragmentation, creating and modifying fragmented tables, and providing privileges for fragmented tables.

For information about how to formulate a fragmentation strategy to reduce data contention and improve query performance, see your *Informix® Performance Guide*.

What is fragmentation?

Fragmentation is a database server feature that allows you to control where data is stored at the table level. Fragmentation enables you to define groups of rows or index keys within a table according to some algorithm or *scheme*. You can store each group or *fragment* (also referred to as a *partition*) in a separate dbspace associated with a specific physical disk. You use SQL statements to create the fragments and assign them to dbspaces.

The scheme that you use to group rows or index keys into fragments is called the *distribution scheme*. The distribution scheme and the set of dbspaces in which you locate the fragments together make up the *fragmentation strategy*. The decisions that you must make to formulate a fragmentation strategy are explained in your *Informix® Performance Guide*.

After you decide whether to fragment table rows, index keys, or both, and you decide how the rows or keys should be distributed over fragments, you decide on a scheme to implement this distribution. For a description of the distribution schemes that Informix® database servers support, see Distribution schemes for table fragmentation on page 51.

When you create fragmented tables and indexes, the database server stores the location of each table and index fragment with other related information in the system catalog table named *sysfragments*. You can use this table to access information about your fragmented tables and indexes. If you use a user-defined routine as part of the fragmentation expression, that information is recorded in **sysfragexprudrdep**. For a description of the information that these system catalog tables contain, see the *Informix® Guide to SQL: Reference*.

From the perspective of an end user or client application, a fragmented table is identical to a nonfragmented table. Client applications do not require any modifications to allow them to access the data in fragmented tables.

For some distribution schemes, the database server has information about which fragments contain which data, so it can route client requests for data to the correct fragment without accessing irrelevant fragments. (The database server cannot route client requests for data to the correct fragment for round-robin and some expression-based distribution schemes.) For more information, see Distribution schemes for table fragmentation on page 51.)

Why use fragmentation?

Consider fragmenting your tables if improving at least one of the following is your goal:

- Single-user response time
- Concurrency
- Availability

- Backup-and-restore characteristics
- Loading of data

Each of the preceding goals has its own implications for the fragmentation strategy that you ultimately implement. Your primary fragmentation goal determines, or at least influences, how you implement your fragmentation strategy. When you decide whether to use fragmentation to meet any of the preceding goals, keep in mind that fragmentation requires some additional administration and monitoring activity.

For more information about the preceding goals and how to plan a fragmentation strategy, see your *Informix® Performance Guide*.

Whose responsibility is fragmentation?

Some overlap exists between the responsibilities of the database server administrator and those of the database administrator (DBA) with respect to fragmentation. The DBA creates the database schema, which can include table fragmentation. The database server administrator, however, is responsible for allocating the disk space in which the fragmented tables will be located. Because neither of these responsibilities can be performed in isolation from the other, to implement fragmentation requires a cooperative effort between the DBA and the database server administrator. This manual describes only those tasks that the DBA performs to implement a fragmentation strategy. For information about the tasks the database server administrator performs to implement a fragmentation strategy, see your *Informix® Administrator's Guide* and *Informix® Performance Guide*.

Fragmentation and logging

Fragmented tables can belong to either a logging database or a nonlogging database. As with nonfragmented tables, if a fragmented table is part of a nonlogging database, a potential for data inconsistencies arises if a failure occurs.

Distribution schemes for table fragmentation

A *distribution scheme* is a method that the database server uses to distribute rows or index entries to fragments. Informix® database servers support the following distribution schemes:

Expression-based

This distribution scheme puts rows that contain specified values in the same fragment. You specify a *fragmentation expression* that defines criteria for assigning a set of rows to each fragment, either as a range rule or some arbitrary rule. You can specify a *remainder fragment* that holds all rows that do not match the criteria for any other fragment, although a remainder fragment reduces the efficiency of the expression-based distribution scheme.

Round-robin

This distribution scheme places rows one after another in fragments, rotating through the series of fragments to distribute the rows evenly. The database server defines the rule internally.

For INSERT statements, the database server uses a hash function on a random number to determine the fragment in which to place the row. For INSERT cursors, the database server places the first row in a random fragment, the second in the next sequential fragment, and so on. If one of the fragments is full, it is skipped.

For complete descriptions of the SQL syntax you use to specify a distribution scheme, see the CREATE TABLE and CREATE INDEX statements in the *Informix® Guide to SQL: Syntax*. For an explanation about the performance aspects of fragmentation, see your *Informix® Performance Guide*.

Expression-based distribution scheme

To specify an expression-based distribution scheme, use the FRAGMENT BY EXPRESSION clause of the CREATE TABLE or CREATE INDEX statement. The following example includes a FRAGMENT BY EXPRESSION clause to create a fragmented table with an expression-based distribution scheme:

```
CREATE TABLE accounts (id_num INT, name char(15))
FRAGMENT BY EXPRESSION
id_num <= 100 IN dbspace_1,
id_num <100 AND id_num <= 200 IN dbspace_2,
id_num > 200 IN dbspace_3
```

When you use the FRAGMENT BY EXPRESSION clause of the CREATE TABLE statement to create a fragmented table, you must supply one condition for each fragment of the table that you are creating.

You can define *range rules* or *arbitrary rules* that indicate to the database server how rows are to be distributed to fragments. The following sections describe the different types of expression-based distribution schemes.

Range rule

A range rule uses SQL relational and logical operators to define the boundaries of each fragment in a table. A range rule can contain the following restricted set of operators:

- The relational operators >, <, >=, <=
- The logical operators AND and OR
- Algebraic expressions including built-in functions

A range rule can be based on a simple algebraic expression as shown in the following example. In this example, the expression is a simple reference to a column.

```
FRAGMENT BY EXPRESSION
id_num > 0  AND id_num <= 20 IN dbsp1,
id_num > 20 AND id_num <= 40 IN dbsp2,
id_num > 40 IN dbsp3
```

The expression in a range rule can be a conjunction or disjunction of more algebraic expressions. The next example shows two algebraic expressions used to define two sets of ranges. The first set of ranges is based on the algebraic expression: "YEAR(Died) - YEAR(Born)"; the second set of ranges is based on "MONTH(Born)."

```
FRAGMENT  BY  EXPRESSION
YEAR(Died) – YEAR(Born) < 21 AND MONTH(Born) >= 1  AND MONTH(Born) < 4 IN dbsp1,
YEAR(Died) – YEAR(Born) < 40 AND MONTH(Born) >= 4  AND MONTH(Born) <  7 IN dbsp2,
```

Arbitrary rule

An arbitrary rule uses SQL relational and logical operators. Unlike range rules, arbitrary rules allow you to use any relational operator and any logical operator to define the rule. In addition, you can reference any number of table columns in the rule. Arbitrary rules typically include the use of the OR logical operator to group data, as the following example shows:

```
FRAGMENT BY EXPRESSION
zip_num = 95228 OR zip_num = 95443 IN dbsp2,
zip_num = 91120 OR zip_num = 92310 IN dbsp4,
REMAINDER IN dbsp5
```

Use the MOD function

You can use the MOD function in a FRAGMENT BY EXPRESSION clause to map each row in a table to a set of integers (hash values). The database server uses these values to determine in which fragment it will store a given row. The following example shows how you might use the MOD function in an expression-based distribution scheme:

```
FRAGMENT BY EXPRESSION
MOD(id_num, 3) = 0 IN dbsp1,
MOD(id_num, 3) = 1 IN dbsp2,
MOD(id_num, 3) = 2 IN dbsp3
```

Insert and update rows

When you insert or update a row, the database server evaluates fragment expressions, in the order specified, to see if the row belongs in any of the fragments. If so, the database server inserts or updates the row in one of the fragments. If the row does not belong in any of the fragments, the row is put into the fragment that the remainder clause specified. If the distribution scheme does not include a remainder clause, and the row does not match the criteria for any of the existing fragment expressions, the database server returns an error.

Round-robin distribution scheme

To specify a round-robin distribution scheme, use the FRAGMENT BY ROUND ROBIN clause of the CREATE TABLE statement. The following statement illustrates a fragmented table with a round-robin distribution scheme:

```
CREATE TABLE account_2
   ...
   ...
FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3
```

When the database server receives a request to insert a number of rows into a table that uses round-robin distribution, it distributes the rows in such a way that the number of rows in each of the fragments remains approximately the same (±100).

Round-robin distributions are also called *even distributions* because information is distributed evenly among the fragments. To that end, a newly added round-robin fragment will be favored exclusively by inserts and loads until it no longer has the fewest number of rows among the table's fragments.

⚠️ **Important:** You can use the round-robin distribution scheme only for table fragmentation. You cannot fragment an index with this distribution scheme.

Create a fragmented table

This section explains how to use SQL statements to create and manage fragmented tables. You can fragment a table at the same time that you create it, or you can fragment existing nonfragmented tables. An overview of both alternatives is given in the following sections. For the complete syntax of the SQL statements that you use to create fragmented tables, see the *Informix® Guide to SQL: Syntax*.

Before you create a fragmented table, you must decide on an appropriate fragmentation strategy. For information about how to formulate a fragmentation strategy, see your *Informix® Performance Guide*.

Create a new fragmented table

To create a fragmented table, use the FRAGMENT BY clause of the CREATE TABLE statement.

Suppose that you want to create a fragmented table similar to the orders table of the **stores_demo** database. You decide on a round-robin distribution scheme with three fragments and consult with your database server administrator to set up three dbspaces, one for each of the fragments: dbspace1, dbspace2, and dbspace3. The following SQL statement creates the fragmented table:

```
CREATE TABLE my_orders (
   order_num      SERIAL(1001),
   order_date     DATE,
   customer_num   INT,
   ship_instruct  CHAR(40),
   backlog        CHAR(1),
   po_num         CHAR(10),
   ship_date      DATE,
   ship_weight    DECIMAL(8,2),
   ship_charge    MONEY(6),
   paid_date      DATE,
   PRIMARY KEY (order_num),
   FOREIGN KEY (customer_num) REFERENCES customer(customer_num))
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3
```

Suppose you want to create multiple tables that are fragmented by round robin and that you want the number of fragments to increase automatically as the tables grow. You set the AUTOLOCATE configuration parameter or session environment variable to the number of initial fragments to create. Tables that do not include the FRAGMENT BY clause in the CREATE TABLE statement are fragmented by round-robin by default into dbspaces that are chosen by the server. By default, all dbspaces are available, but you can control the list of available dbspaces.

You might decide instead to create the table with expression-based fragmentation. Suppose that your **my_orders** table has 30,000 rows, and you want to distribute rows evenly across three fragments stored in dbspace1, dbspace2, and dbspace3. The following statement shows how you might use the order_num column to define an expression-based fragmentation strategy:

```
CREATE TABLE my_orders (order_num SERIAL, ...)
   FRAGMENT BY EXPRESSION
      order_num < 10000 IN dbspace1,
      order_num >= 10000 and order_num < 20000 IN dbspace2,
      order_num >= 20000 IN dbspace3
```

Create a fragmented table from nonfragmented tables

You might be required to convert nonfragmented tables into fragmented tables in the following circumstances:

- You have an application-implemented version of table fragmentation.

  You will probably want to convert several small tables into one large fragmented table. The following section tells you how to proceed when this is the case. Follow the instructions in the section .
- You have an existing large table that you want to fragment.

  Follow the instructions in the section .

Remember: Before you perform the conversion, you must set up an appropriate number of dbspaces to contain the newly created fragmented tables.

More than one nonfragmented table

You can combine two or more nonfragmented tables into a single fragmented table. The nonfragmented tables must have identical table structures and must be stored in separate dbspaces. To combine nonfragmented tables, use the ATTACH clause of the ALTER FRAGMENT statement.

For example, suppose that you have three nonfragmented tables, account1, account2, and account3, and that you store the tables in dbspaces dbspace1, dbspace2, and dbspace3, respectively. All three tables have identical structures, and you want to combine the three tables into one table that is fragmented by the expression on the common column acc_num.

You want rows with acc_num less than or equal to 1120 to be stored in dbspace1. Rows with acc_num greater than 1120 but less than or equal to 2000 are to be stored in dbspace2. Finally, rows with acc_num greater than 2000 are to be stored in dbspace3.

To fragment the tables with this fragmentation strategy, execute the following SQL statement:

```
ALTER FRAGMENT ON TABLE tab1 ATTACH
   tab1 AS acc_num <= 1120,
   tab2 AS acc_num >  1120 and acc_num <= 2000,
   tab3 AS acc_num > 2000;
```

The result is a single table, tab1. The other tables, tab2 and tab3, were consumed and no longer exist.

For information about how to use the ATTACH and DETACH clauses of the ALTER FRAGMENT statement to improve performance, see your *Informix® Performance Guide*.

Use a single nonfragmented table

To create a fragmented table from a nonfragmented table, use the INIT clause of the ALTER FRAGMENT statement. For example, suppose you want to convert the table orders to a table fragmented by round-robin. The following SQL statement performs the conversion:

```
ALTER FRAGMENT ON TABLE orders INIT
    FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

Any existing indexes on the nonfragmented table become fragmented with the same fragmentation strategy as the table.

Rowids in a fragmented table

The term *rowid* refers to an integer that defines the physical location of a row. The rowid of a row in a nonfragmented table is a unique and constant value. Rows in fragmented tables, in contrast, are not assigned a rowid.

⚠️ **Important:** Use primary keys as a method of access in your applications rather than rowids. Because primary keys are defined in the ANSI specification of SQL, using primary keys to access data makes your applications more portable.

To accommodate applications that must reference a rowid for a fragmented table, you can explicitly create a rowid column for a fragmented table. However, you cannot use the WITH ROWIDS clause for typed tables.

To create the rowid column, use the following SQL syntax:

- The WITH ROWIDS clause of the CREATE TABLE statement
- The ADD ROWIDS clause of the ALTER TABLE statement
- The INIT clause of the ALTER FRAGMENT statement

When you create the rowid column, the database server takes the following actions:

- Adds the 4-byte unique value to each row in the table
- Creates an internal index that it uses to access the data in the table by rowid
- Inserts a row in the **sysfragments** system catalog table for the internal index

Fragment smart large objects

You can specify multiple sbspaces in the PUT clause of the CREATE TABLE statement to achieve round-robin fragmentation of smart large objects on a column. If you specify multiple sbspaces for a CLOB or BLOB column, the database server distributes the smart large objects for the column to the specified sbspaces in round-robin fashion. Given the following CREATE TABLE statement, the database server can distribute large objects from the **cat_photo** column to **sbcat1**, **sbcat2**, and **sbcat3** in round-robin fashion.

```
CREATE TABLE catalog (
   catalog_num  SERIAL,
   stock_num    SMALLINT,
   manu_code    CHAR(3),
   cat_descr    LVARCHAR,
   cat_photo    BLOB)
PUT cat_photo in (sbcat1, sbcat2, sbcat3;
```

Modify fragmentation strategies

You can make two general types of modifications to a fragmented table. The first type consists of the modifications that you can make to a nonfragmented table. Such modifications include adding a column, dropping a column, changing a column data type, and so on. For these modifications, use the ALTER TABLE statements that you would normally use on a nonfragmented table. The second type of modification consists of changes to a fragmentation strategy. This section explains how to use SQL statements to modify fragmentation strategies.

At times, you might be required to alter a fragmentation strategy after you implement fragmentation. Most frequently, you will be required to modify your fragmentation strategy when you use fragmentation with intraquery or interquery parallelization. Modifying your fragmentation strategy in these circumstances is one of several ways you can improve the performance of your database server system.

Reinitialize a fragmentation strategy

You can use the ALTER FRAGMENT statement with an INIT clause to define and initialize a new fragmentation strategy on a nonfragmented table or convert an existing fragmentation strategy on a fragmented table. You can also use the INIT clause to change the order of evaluation of fragment expressions.

The following example shows how you might use the INIT clause to reinitialize a fragmentation strategy completely.

Suppose that you initially create the following fragmented table:

```
CREATE TABLE account (acc_num INTEGER, ...)
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 in dbspace1,
      acc_num > 1120 and acc_num < 2000 in dbspace2,
      REMAINDER IN dbspace3;
```

Suppose that after several months of operation with this distribution scheme, you find that the number of rows in the fragment contained in dbspace2 is twice the number of rows that the other two fragments contain. This imbalance causes the disk that contains dbspace2 to become an I/O bottleneck.

To remedy this situation, you decide to modify the distribution so that the number of rows in each fragment is approximately even. You want to modify the distribution scheme so that it contains four fragments instead of three fragments. A new dbspace, dbspace2a, is to contain the new fragment that stores the first half of the rows that previously were contained in dbspace2. The fragment in dbspace2 contains the second half of the rows that it previously stored.

To implement the new distribution scheme, first create the dbspace dbspace2a and then execute the following statement:

```
ALTER FRAGMENT ON TABLE account INIT
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 in dbspace1,
      acc_num >  1120 and acc_num <= 1500 in dbspace2a,
      acc_num >  1500 and acc_num < 2000 in dbspace2,
      REMAINDER IN dbspace3;
```

As soon as you execute this statement, the database server discards the old fragmentation strategy, and the rows that the table contains are redistributed according to the new fragmentation strategy.

You can also use the INIT clause of ALTER FRAGMENT to perform the following actions:

  • Convert a single nonfragmented table into a fragmented table
  • Convert a fragmented table into a nonfragmented table
  • Convert a table fragmented by any strategy to any other fragmentation strategy

For more information, see the ALTER FRAGMENT statement in the *Informix® Guide to SQL: Syntax*.

Modify fragmentation strategies

You can use the ADD, DROP, and MODIFY clauses to change the fragmentation strategy on a table or index. For syntax information about these options, see the ALTER FRAGMENT statement in the *Informix® Guide to SQL: Syntax*.

The ADD clause

When you define a fragmentation strategy, you may be required to add one or more fragments. You can use the ADD clause of the ALTER FRAGMENT statement to add a new fragment to a table. Suppose that you want to add a fragment to a table that you create with the following statement:

```
CREATE TABLE sales (acc_num INT, ...)
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

To add a new fragment dbspace4 to the table **sales**, execute the following statement:

```
ALTER FRAGMENT ON TABLE sales ADD dbspace4;
```

If the fragmentation strategy is expression based, the ADD clause of ALTER FRAGMENT contains options to add a dbspace before or after an existing dbspace.

The DROP clause

When you define a fragmentation strategy, you must drop one or more fragments. With , you can use the DROP clause of the ALTER FRAGMENT ON TABLE statement to drop a fragment from a table. Suppose you want to drop a fragment from a table that you create with the following statement:

```
CREATE TABLE sales (col_a INT), ...)
   FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3;
```

The following ALTER FRAGMENT statement uses a DROP clause to drop the third fragment **dbspace3** from the **sales** table:

```
ALTER FRAGMENT ON TABLE sales DROP dbspace3;
```

When you issue this statement, all the rows in dbspace3 are moved to the remaining dbspaces, dbspace1 and dbspace2.

The MODIFY clause

Use the ALTER FRAGMENT statement with the MODIFY clause to modify one or more of the expressions in an existing fragmentation strategy.

Suppose that you initially create the following fragmented table:

```
CREATE TABLE account (acc_num INT, ...)
   FRAGMENT BY EXPRESSION
      acc_num <= 1120 IN dbspace1,
      acc_num > 1120 AND acc_num < 2000 IN dbspace2,
      REMAINDER IN dbspace3;
```

When you execute the following ALTER FRAGMENT statement, you ensure that no account numbers with a value less than or equal to zero are stored in the fragment that dbspace1 contains:

```
ALTER FRAGMENT ON TABLE account
   MODIFY dbspace1 TO acc_num > 0 AND acc_num <=1120;
```

You cannot use the MODIFY clause to alter the number of fragments that your distribution scheme contains. Use the INIT or ADD clause of ALTER FRAGMENT instead.

Grant and revoke privileges on fragments

You must have a strategy to control data distribution if you want to grant useful fragment privileges. One effective strategy is to fragment data records by expression. The round-robin data-record distribution strategy, however, is not a useful strategy because each new data record is added to the next fragment. A round-robin distribution nullifies any clean method of tracking data distribution and therefore eliminates any real use of fragment authority. Because of this difference between expression-based distribution and round-robin distribution, the GRANT FRAGMENT and REVOKE FRAGMENT statements apply only to tables that have expression-based fragmentation.

When you create a fragmented table, no default fragment authority exists. Use the GRANT FRAGMENT statement to grant insert, update, or delete authority on one or more of the fragments. If you want to grant all three privileges simultaneously,

use the ALL keyword of the GRANT FRAGMENT statement. However, you cannot grant fragment privileges by merely naming the table that contains the fragments. You must name the specific fragments.

When you want to revoke insert, update, or delete privileges, use the REVOKE FRAGMENT statement. This statement revokes privileges from one or more users on one or more fragments of a fragmented table. If you want to revoke all privileges that currently exist for a table, you can use the ALL keyword. If you do not specify any fragments in the command, the permissions being revoked apply to all fragments in the table that currently have permissions.

For more information, see the GRANT FRAGMENT, REVOKE FRAGMENT, and SET statements in the *Informix® Guide to SQL: Syntax*.

Grant and limit access to your database

This chapter describes how you can control access to your database. In some databases, all data is accessible to every user. In others, some users are denied access to some or all the data.

Use SQL to restrict access to data

You can restrict access to data at the following levels:

- You can use the GRANT and REVOKE statements to give or deny access to the database or to specific tables, and you can control the kinds of uses that people can make of the database.
- You can use the CREATE PROCEDURE or CREATE FUNCTION statement to write and compile a user-defined routine, which controls and monitors the users who can read, modify, or create database tables.
- You can use the CREATE VIEW statement to prepare a restricted or modified view of the data. The restriction can be vertical, which excludes certain columns, or horizontal, which excludes certain rows, or both.
- You can combine GRANT and CREATE VIEW statements to achieve precise control over the parts of a table that a user can modify and with what data.
- You can use the SET ENCRYPTION PASSWORD statement and built-in encryption and decryption functions of SQL to implement column-level encryption of sensitive data. Unauthorized users who succeed in viewing an encrypted character, BLOB, or CLOB column value cannot recover the plain text of your data without the DES or triple-DES encryption key, which is not stored in the database.

Control access to databases

Grant privileges on page 61 contains information about how the normal database-privilege mechanisms are based on the GRANT and REVOKE statements. You can sometimes use the facilities of the operating system, however, as an additional way to control access to a database.

No matter what access controls the operating system gives you, when the contents of an entire database are highly sensitive, you might not want to leave it on a public disk that is fixed to the computer. You can circumvent normal software controls when the data must be secure.

When you or another authorized person is not using the database, it does not have to be available online. You can make it inaccessible in one of the following ways, which have varying degrees of inconvenience:

- Detach the physical medium from the computer and take it away. If the disk itself is not removable, the disk drive might be removable.
- Copy the database directory to tape and take possession of the tape.
- Use an encryption utility to copy the database files. Keep only the encrypted version.

Important: In the latter two cases, after making the copies, you must remember to erase the original database files with a program that overwrites an erased file with NULL data.

Instead of removing the entire database directory, you can copy and then erase the files that represent individual tables. Do not overlook the fact that index files contain copies of the data from the indexed column or columns. Remove and erase the index and table files.

Grant privileges

The authorization to use a database is called an *access privilege*. For example, the authorization to use a database is called the *Connect privilege*; authorization to insert a row into a table is called the *Insert privilege*. Use the GRANT statement to grant privileges on a database, table, view, or procedure, or to grant a role to a user or another role. Use the REVOKE statement to revoke privileges on a database or database object, or to revoke a role from a user or from another role.

A *role* is a classification of access privileges that the DBA assigns, such as **payroll**. After a role is created with the CREATE ROLE statement, the DBA can use the GRANT statement to assign access privileges to the role, and to assign the role to individual users (or to other roles), so that users with similar work tasks can hold the set of access privileges that their work tasks require. By assigning privileges to roles and roles to users, you can simplify the management of privileges. See also External routines on page 70 and Roles on page 71 for additional information about the role of roles in managing access privileges.

The following groups of privileges control the actions a user can perform on data and on database objects:

- Database-level privileges
- Ownership privileges
- Table-level privileges
- Column-level privileges
- Type-level privileges
- Routine-level privileges
- Language-level privileges
- Automating privileges

For the syntax of the GRANT and REVOKE statements, see the *Informix® Guide to SQL: Syntax*.

Database-level privileges

The three levels of database privileges provide an overall means of controlling who accesses a database. Only individual users, not roles, can hold database-level privileges.

Connect privilege

The least of the privilege levels is Connect, which gives a user the basic ability to query and modify tables.

Users with the Connect privilege can perform the following functions:

- Execute the SELECT, INSERT, UPDATE, and DELETE statements, provided that they have the necessary table-level privileges.
- Execute an SPL routine, provided that they have the necessary table-level privileges.
- Create views, provided that they are permitted to query the tables on which the views are based.
- Create temporary tables and create indexes on the temporary tables.

Before users can access a database, they must have the Connect privilege. Ordinarily, in a database that does not contain highly sensitive or private data, you give the GRANT CONNECT TO PUBLIC privilege shortly after you create the database.

If you do not grant the Connect privilege to PUBLIC, the only users who can access the database through the database server are those to whom you specifically grant the Connect privilege. If limited users should have access, this privilege lets you provide it to them and deny it to all others.

Related information

Users and the public

Privileges are granted to single users by name or to all users under the name of PUBLIC. Any privileges granted to PUBLIC serve as default privileges.

Before executing a statement, the database server determines whether a user has the necessary privileges. The information is in the system catalog. For more information, see .

The database server looks first for privileges that are granted specifically to the requesting user. If it finds such a grant, it uses that information. It then checks to see if less restrictive privileges were granted to PUBLIC. If they were, the database server uses the less restrictive privileges. If no grant has been made to that user, the database server looks for privileges granted to PUBLIC. If it finds a relevant privilege, it uses that one.

Thus, to set a minimum level of privilege for all users, grant privileges to PUBLIC. You can override that, in specific cases, by granting higher individual privileges to users.

Resource privilege

The Resource privilege carries the same authorization as the Connect privilege. In addition, users with the Resource privilege can create new, permanent tables, indexes, and SPL routines, thus permanently allocating disk space.

Related information

Database-administrator privilege

The highest level of database privilege is database administrator, or DBA. When you create a database, you are automatically the DBA.

Holders of the DBA privilege can perform the following functions:

- Execute the DROP DATABASE, START DATABASE, and ROLLFORWARD DATABASE statements.
- Drop or alter any object regardless of who owns it.
- Create tables, views, and indexes to be owned by other users.
- Grant database privileges, including the DBA privilege, to another user.

Only the user **informix** can modify system catalog tables directly. If you are user **informix**, strongly recommends that you not modify the contents or schema of any system catalog table, because such actions can affect the integrity of the database.

Related information

Ownership rights

The database, and every table, view, index, procedure, and synonym in it, has an owner. The owner of an object is usually the person who created it, although a user with the DBA privilege can create objects to be owned by others.

The owner of a database object has all rights to that object and can alter or drop it without additional privileges.

Table-level privileges

You can apply seven privileges, table by table, to allow nonowners the privileges of owners. Four of them, the Select, Insert, Delete, and Update privileges, control DML access to data in the table. The Index privilege controls index creation. The Alter privilege gives authorization to change the table definition. The References privilege gives authorization to specify referential constraints on a table.

In an ANSI-compliant database, only the table owner has any privileges. In other databases, the database server, as part of creating a table, automatically grants to PUBLIC all table privileges except Alter and References, unless the **NODEFDAC** environment variable has been set to 'yes' to withhold all table privileges from PUBLIC. When you allow the database server to automatically grant all table privileges to PUBLIC, a newly created table is accessible to any user with the Connect privilege. If this is not what you want (if users exist with the Connect privilege who should not be able to access this table), you must revoke all privileges on the table from PUBLIC after you create the table.

Access privileges

Four privileges govern how users can access a table. As the owner of the table, you can grant or withhold the following privileges independently:

- Select allows selection, including selecting into temporary tables.
- Insert allows a user to add new rows.
- Update allows a user to modify existing rows.
- Delete allows a user to delete rows.

The Select privilege is necessary for a user to retrieve the contents of a table. However, the Select privilege is not a precondition for the other privileges. A user can have Insert or Update privileges without having the Select privilege.

For example, your application might have a usage table. Every time a certain program is started, it inserts a row into the usage table to document that it was used. Before the program terminates, it updates that row to show how long it ran and perhaps to record counts of work its user performs.

If you want any user of the program to be able to insert and update rows in this usage table, grant Insert and Update privileges on it to PUBLIC. However, you might grant the Select privilege to only a few users.

Privileges in the system catalog tables

Database-level and table-level privileges are recorded in the system catalog tables. Any user with the Connect privilege can query the system catalog tables to determine what privileges are granted and to whom.

Database-level privileges and roles are recorded in the **sysusers** system catalog table, in which the primary key is the **username** column, and the **usertype** column contains a single character C (for Connect), R (for Resource), or D (for DBA) that specifies the highest database-level privilege that **username** holds, or G if **username** is the authorization identifier of a role. The last column, **defrole**, stores the default role if **username** holds a default role. (Neither a default role nor database- level privileges can be granted to a role, but a role can hold other access privileges, such as table-level privileges, and a role can be granted a non-default role.) The **username** in the row that shows the highest database-level privilege held by the PUBLIC group is public.

Table-level privileges are recorded in **systabauth** system catalog table, which uses a composite primary key of the table number, grantor, and grantee. In the **tabauth** column, the privileges are encoded in the list as follows.

Code

Meaning

s

unconditional select

u

update

-

ungranted privileges

i

insert

d

delete

x

index

a

alter

r

references

A hyphen means an ungranted privilege, so that a grant of all privileges is shown as `su-idxar`, and `-u------` shows a grant of only Update. The code letters are normally lowercase, but they are uppercase when the keywords WITH GRANT OPTION are used in the GRANT statement.

When an asterisk (*) appears in the third position, some column-level privilege exists for that table and grantee. The specific privilege is recorded in **syscolauth**. Its primary key is a composite of the table number, the grantor, the grantee, and the column number. The only attribute is a three-letter list that shows the type of privilege: `s`, `u`, or `r`.

Index, alter, and references privileges

The Index privilege permits its holder to create and alter indexes on the table. The Index privilege, similar to the Select, Insert, Update, and Delete privileges, is granted automatically to PUBLIC when you create a table.

You can grant the Index privilege to anyone, but to exercise the privilege, the user must also hold the Resource database privilege. So, although the Index privilege is granted automatically (except in ANSI-compliant databases), users who have only the Connect privilege to the database cannot exercise their Index privilege. Such a limitation is reasonable because an index can fill a large amount of disk space.

The Alter privilege permits its holder to use the ALTER TABLE statement on the table, including the power to add and drop columns, reset the starting point for SERIAL columns, and so on. You should grant the Alter privilege only to users who understand the data model well and whom you trust to exercise their power carefully.

The References privilege allows you to impose referential constraints on a table. As with the Alter privilege, you should grant the References privilege only to users who understand the data model well.

Under privileges for typed tables

You can grant or revoke the Under privilege to control whether users can use a typed table as a supertable in an inheritance hierarchy. The Under privilege is granted to PUBLIC automatically when a table is created (except in ANSI-compliant databases). In an ANSI-compliant database, the Under privilege on a table is granted to the owner of the table. To restrict which users can define a table as a supertable in an inheritance hierarchy, you must first revoke the Under privilege for PUBLIC and then specify the users to whom you want to grant the Under privilege. For example, to specify that only a limited group of users can use the **employee** table as a supertable in an inheritance hierarchy, you might execute the following statements:

```
REVOKE UNDER ON employee
   FROM PUBLIC;

GRANT UNDER ON employee
   TO johns, cmiles, paulz
```

For information about how to use the UNDER clause to create tables in an inheritance hierarchy, see Table inheritance on page 121.

Privileges on table fragments

Use the GRANT FRAGMENT statement to grant insert, update, and delete privileges on individual fragments of a fragmented table. The GRANT FRAGMENT statement is valid only for tables that are fragmented with expression-based distribution schemes.

Suppose you create a **customer** table that is fragmented by expression into three fragments, which are located in the dbspaces **dbsp1**, **dbsp2**, and **dbsp3**. The following statement shows how to grant insert privileges on the first two fragments only (**dbsp1** and **dbsp2**) to users **jones**, **reed**, and **mathews**.

```
GRANT FRAGMENT INSERT ON customer (dbsp1, dbsp2)
   TO jones, reed, mathews
```

To grant privileges on all fragments of a table, use the GRANT statement or the GRANT FRAGMENT statement.

For information about the GRANT FRAGMENT and REVOKE FRAGMENT statements, see the *Informix® Guide to SQL: Syntax*.

Column-level privileges

You can qualify the Select, Update, and References privileges with the names of specific columns. Naming specific columns allows you to grant specific access to a table. You can permit a user to see only certain columns, to update only certain columns, or to impose referential constraints on certain columns.

You can use the GRANT and REVOKE statements to grant or restrict access to table data. This feature solves the problem that only certain users should know the salary, performance review, or other sensitive attributes of an employee. Suppose a table of employee data is defined as the following example shows:

```
CREATE TABLE hr_data
   (
   emp_key INTEGER,
   emp_name CHAR(40),
   hire_date DATE,
   dept_num SMALLINT,
   user-id CHAR(18),
   salary DECIMAL(8,2)
   performance_level CHAR(1),
   performance_notes TEXT
   )
```

Because this table contains sensitive data, you execute the following statement immediately after you create it:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

For selected persons in the Human Resources department, and for all managers, execute the following statement:

```
GRANT SELECT ON hr_data TO harold_r
```

In this way, you permit certain users to view all columns. (The final section of this chapter contains information about a way to limit the view of managers to their employees only.) For the first-line managers who carry out performance reviews, you can execute a statement such as the following one:

```
GRANT UPDATE (performance_level, performance_notes)
   ON hr_data TO wallace_s, margot_t
```

This statement permits the managers to enter their evaluations of their employees. You would execute a statement such as the following one only for the manager of the Human Resources department or whomever is trusted to alter salary levels:

```
GRANT UPDATE (salary) ON hr_data to willard_b
```

For the clerks in the Human Resources department, you can execute a statement such as the following one:

```
GRANT UPDATE (emp_key, emp_name, hire_date, dept_num)
   ON hr_data TO marvin_t
```

This statement gives certain users the ability to maintain the nonsensitive columns but denies them authorization to change performance ratings or salaries. The person in the MIS department who assigns computer user IDs is the beneficiary of a statement such as the following one:

```
GRANT UPDATE (user_id) ON hr_data TO eudora_b
```

On behalf of all users who are allowed to connect to the database, but who are not authorized to see salaries or performance reviews, execute statements such as the following one to permit them to see the nonsensitive data:

```
GRANT SELECT (emp_key, emp_name, hire_date, dept_num, user-id)
   ON hr_data TO george_b, john_s
```

These users can perform queries such as the following one:

```
SELECT COUNT(*) FROM hr_data WHERE dept_num IN (32,33,34)
```

However, any attempt to execute a query such as the following one produces an error message and no data:

```
SELECT performance_level FROM hr_data
   WHERE emp_name LIKE '*Smythe'
```

Type-level privileges

supports user-defined data types (UDTs). When a user-defined data type is created, only the DBA or owner of the data type can grant or revoke type-level privileges that control who can use the UDT. supports the two type-level privileges.

Usage privileges for user-defined types

To control who can use an opaque type, distinct type, or named row type, specify the Usage privilege on the data type. The Usage privilege allows the DBA or owner of the type to restrict a user's ability to assign a data type to a column, program variable (or table or view for a named row type), or assign a cast to the data type. The Usage privilege is granted to PUBLIC automatically when a data type is created (except in ANSI-compliant databases). In an ANSI-compliant database, the Usage privilege on a data type is granted to the owner of the data type.

To limit who can use an opaque, distinct, or named row type, you must first revoke the Usage privilege for PUBLIC and then specify the names of the users to whom you want to grant the Usage privilege. For example, to limit the use of a data type named **circle** to a group of users, you might execute the following statements:

```
REVOKE USAGE ON circle
   FROM PUBLIC;

GRANT USAGE ON circle
   TO dawns, stevep, terryk, camber;
```

Related information

Select a name for a named row type on page 110

Under privileges for named row types

For named row types, you can grant or revoke the Under privilege, which controls whether users can assign a named row type as the supertype of another named row type in an inheritance hierarchy. The Under privilege is granted to PUBLIC automatically when a named row type is created (except in ANSI-compliant databases). In an ANSI-compliant database, the Under privilege on a named row type is granted to the owner of the type.

To restrict certain users' ability to define a named row type as a supertype in an inheritance hierarchy, you must first revoke the Under privilege for PUBLIC and then specify the names of the users to whom you want to grant the Under privilege. For example, to specify that only a limited group of users can use the named row type **person_t** as a supertype in an inheritance hierarchy, you might execute the following statements:

```
REVOKE UNDER ON person_t
   FROM PUBLIC;

GRANT UNDER ON person_t
   TO howie, jhana, alison
```

For information about how to use the UNDER clause to create named row types in an inheritance hierarchy, see Type inheritance on page 116.

Routine-level privileges

You can apply the Execute privilege on a user-defined routine (UDR) to authorize nonowners to execute the UDR. If you create a UDR in a database that is not ANSI-compliant, the default routine-level privilege is PUBLIC; you are not required to grant the Execute privilege to specific users unless you have first revoked it. If you create a routine in an ANSI-compliant database, no other users have the Execute privilege by default; you must grant specific users the Execute privilege. The following example grants the Execute privilege to the user **orion** so that **orion** can use the UDR that is named **read_address**:

```
GRANT EXECUTE ON ROUTINE read_address TO orion;
```

The **sysprocauth** system catalog table records routine-level privileges. The **sysprocauth** system catalog table uses a primary key of the routine number, grantor, and grantee. In the **procauth** column, the execute privilege is indicated by a lowercase *"e"*. If the execute privilege was granted with the WITH GRANT option, the privilege is represented by an uppercase *"E"*.

For more information about routine-level privileges, see the *Informix® Guide to SQL: Tutorial*.

Language-level privileges

supports UDRs written in the built-in Stored Procedure Language (SPL) and also UDRs (referred to as *external routines*) that are written the C and Java™ languages. To create any UDR, a user must hold the Resource privilege (or else DBA privilege) on the database. In addition, to create a UDR, the user must also receive the Usage privilege on the programming language from the corresponding GRANT statement:

- `GRANT USAGE ON LANGUAGE C` for C routines
- `GRANT USAGE ON LANGUAGE JAVA` for Java™ I routines
- `GRANT USAGE ON LANGUAGE SPL` for SPL routines

In addition to holding the required language-level privileges, if the IFX_EXTEND_ROLE configuration parameter has been enabled (either by default, or by being set to `1` or to `ON`), only users to whom the DBSA has granted the built-in EXTEND role can create, alter, or drop external routines.

SPL routines

By default, language usage privilege on SPL is granted to user **informix** and to users who hold the DBA privilege. Only user **informix**, however, can grant language usage privileges to other users. Users with the DBA privilege hold language usage privileges, but cannot grant these privileges to other users. Usage privilege to create SPL routines is granted to PUBLIC by default.

The following statement shows how user **informix** might revoke from PUBLIC but grant to users **mays**, **jones**, and **freeman** permission to create UDRs in SPL:

```
REVOKE USAGE ON LANGUAGE SPL FROM PUBLIC
GRANT USAGE ON LANGUAGE SPL TO mays, jones, freeman
```

Suppose the default Usage privileges on an SPL routine have been revoked from PUBLIC. The following statement shows how a user with the DBA privilege might grant Usage privilege to register SPL routines to users **franklin**, **reeves**, and **wilson**:

```
GRANT USAGE ON LANGUAGE SPL TO franklin, reeves, wilson
```

External routines

This release of does not support language-level privileges on external routines that are written in the C or Java™ language. When the IFX_EXTEND_ROLE configuration parameter to ON, however, equivalent functionality is provided through the built-in EXTEND role, which is required for any user to register, drop, or replace a UDR or a DataBlade® module that is written in the C or Java™ language.

Only the database server administrator (DBSA), by default user **informix**, can grant the EXTEND role. In contrast with user-defined role names, built-in roles such as EXTEND and DBSECADM are automatically active, and the privileges conferred by the role cannot be modified. When the EXTEND role is enabled, only users who have been granted the EXTEND role can create or drop a DataBlade® module or an external UDR.

The DBSA also has the option of disabling this restriction by setting the IFX_EXTEND_ROLE configuration parameter to OFF, or to leave it unset. In this case, any user who holds the RESOURCE privilege on the database can create a UDR written in the C or Java™ language.

Automate privileges

This design might seem to force you to execute a tedious number of GRANT statements when you first set up the database. Furthermore, privileges require constant maintenance, as people change jobs. For example, if a clerk in Human Resources is terminated, you want to revoke the Update privilege as soon as possible, otherwise the unhappy employee might execute a statement such as the following one:

```
UPDATE hr_data
    SET (emp_name, hire_date, dept_num) = (NULL, NULL, 0)
```

Less dramatic, but equally necessary, privilege changes are required daily, or even hourly, in any model that contains sensitive data. If you anticipate this requirement, you can prepare some automated tools to help maintain privileges.

Your first step should be to specify privilege classes that are based on the jobs of the users, not on the structure of the tables. For example, a first-line manager requires the following privileges:

- The Select and limited Update privileges on the hypothetical **hr_data** table
- The Connect privilege to this and other databases
- Some degree of privilege on several tables in those databases

When a manager is promoted to a staff position or sent to a field office, you must revoke all those privileges and grant a new set of privileges.

Define the privilege classes you support, and for each class specify the databases, tables, and columns to which you must give access. Then devise two automated routines for each class, one to grant the class to a user and one to revoke it.

Automate with a command script

Your operating system probably supports automatic execution of command scripts. In most operating environments, interactive SQL tools such as DB-Access accept commands and SQL statements to execute from the command line. You can combine these two features to automate privilege maintenance.

The details depend on your operating system and the version of the interactive SQL tool that you are using. You must create a command script that performs the following functions:

- Takes a user ID whose privileges are to be changed as its parameter
- Prepares a file of GRANT or REVOKE statements customized to contain that user ID
- Invokes the interactive SQL tool (such as DB-Access) with parameters that tell it to select the database and execute the prepared file of GRANT or REVOKE statements

In this way, you can reduce the change of the privilege class of a user to one or two commands.

Roles

Another way to avoid the difficulty of changing user privileges on a case-by-case basis is to use roles. The concept of a role in the database environment is similar to the group concept in an operating system. A role is a database feature that lets the DBA standardize and change the access privileges of many users by treating them as members of a class. (User-defined roles cannot be granted the database-level privileges Connect, Resource, or DBA, but roles can hold discretionary access privileges on database objects, including privileges on table objects, on fragments of tables, on user-defined data types, on user-defined routines, and on programming languages.)

For example, if you grant the Connect privilege to the PUBLIC group for each of the databases that handle company news and messages, you can create a role called **news_mes** to which you grant the Insert and Delete privileges on tables in which employees who are granted that role can add or delete rows. When a new employee arrives, you must only add that person to the **news_mes** role. By issuing the `SET ROLE news_mes` statement to enable that role, the new employee acquires the access privileges of the **news_mes** role. (Alternatively, you can define a *user*.**sysdbopen** procedure in each database where those privileges are needed, where *user* is the authorization identifier of the new employee, to execute the `SET ROLE news_mes` statement automatically when the user connects to the database.)

This process also works in reverse. To change the discretionary access privileges of everyone who has been granted the **news_mes** role, use the GRANT or REVOKE statements to change the privileges of that role in each database where the **news_mes** role is defined.

> 📝 **Note:** Access privileges granted to individual users, however, or that users hold as members of the PUBLIC group, are not affected when the same privileges are revoked from a user-defined role that those users hold, or when the role is revoked from them, or when the role is dropped.

Create a role

To start the role creation process, determine the name of the role and the connections and privileges that you want to grant to users who hold that role. Although the connections and privileges are strictly in your domain, you must consider some factors when you declare the name of a new role. For the name of a user-defined role, do not use any of the following SQL keywords, access privileges, or built-in roles:

- ALTER
- C
- CONNECT
- DBA
- DBSECADM
- DEFAULT
- DELETE
- EXECUTE
- EXTEND
- INDEX
- INSERT
- NONE
- NULL
- PUBLIC
- REFERENCES
- RESOURCE
- SELECT
- SETSESSIONAUTH
- SPL
- UPDATE

Because role names are authorization identifiers, rather than SQL identifiers, the maximum length of a role name is 32 bytes.

A role name must be different from existing role names in the same database. A role name must also be different from user names that are known to the operating system, including network users known to the server computer. To make sure that your new role name is unique, check the names of the users in the shared memory structure who are currently using the database, and in the following system catalog tables:

- **sysusers**
- **systabauth**

- **syscolauth**
- **sysfragauth**
- **sysprocauth**
- **sysroleauth**
- **syssecpolicyexemptions**
- **sysxtdtypeauth**

When the situation is reversed and you are adding a user to the database, check that the user name is not the same as any of the existing role names.

After you approve the role name, use the CREATE ROLE statement to create a new role. After the role is created, all privileges for role administration are, by default, given to the DBA.

> ❗ **Important:** The scope of a role is the current database only. When you execute the SET ROLE statement, the specified role takes effect in the current database only. As a security precaution, a user who holds access privileges only through a role cannot access tables in a remote database through a view, trigger, or procedure.

Manipulate user privileges and grant roles to other roles

As DBA, you can use the GRANT statement to grant role privileges to users. You can also give a user the option to grant privileges to other users. Use the WITH GRANT OPTION clause of the GRANT statement to do this.

You can also use the WITH GRANT OPTION clause when granting privileges to roles as in this example:

```
GRANT rol1 TO usr1 WITH GRANT OPTION;
```

The WITH GRANT OPTION clause is valid only for roles (and for access privileges) that you grant to users. The database server issues an error if you include the WITH GRANT OPTION keywords when the TO clause specifies a role, or when it specifies the PUBLIC group.

When you grant role privileges, you can substitute a role name for the user name in the GRANT statement. You can grant a role to another role. For example, say that role A is granted to role B. When a user enables role B, the user gets privileges from both role A and role B.

However, a cycle of role granting cannot be transitive. If role A is granted role B, and role B is granted role C, then granting C to A returns an error.

If you must change privileges, use the REVOKE statement to delete the existing privileges and then use the GRANT statement to add the new privileges.

Enable default roles and non-default roles

After the DBA grants privileges and adds users to a role, there are two possible ways to enable roles.

- The DBSA can specify a *default role* for PUBLIC or for individual users by using the GRANT DEFAULT ROLE statement. This role is automatically activated as the initial role setting when the user connects to the database.
- Any role that a user holds can also be activated when the user specifies that role in the SET ROLE statement.

When a role is enabled, all privileges that have been granted to the role become available, and all privileges explicitly granted to you or to PUBLIC.

Assigning privileges to a role, and then granting that role as the default role to specified users is convenient for sessions in which those users run an application that requires a specific set of access privileges. Use default roles when it is impractical to recompile an application to include GRANT and SET ROLE statements that specifically assign to users the necessary access privileges.

Confirm membership In roles and drop roles

You can find yourself in a situation where you are uncertain which user is included in a role. Perhaps you did not create the role, or the person who created the role is unavailable. Issue queries against the **sysroleauth** and **sysusers** system catalog tables to find who is authorized for which table and how many roles exist.

After you determine which users hold which roles, you might discover that some roles are no longer useful. To remove a role, use the DROP ROLE statement. Before you remove a role, the following conditions must be met:

- Only roles that are listed in the **sysusers** system catalog table as a role can be deleted, but you cannot drop a built-in role (such as NONE or EXTEND).
- You must have DBA privileges, or you must be given the grantable option in the role to drop a role.

Determine current role at runtime

If you experience unexpected errors with a role that was granted appropriate access privileges, make sure that the role was enabled during runtime. To obtain this information while you are connected to the database, you can use the onstat -g sql or onstat -g ses command. To see only your own current role, use the **CURRENT_ROLE** operator of SQL. To see your default role, use the **DEFAULT_ROLE** operator of SQL.

Use SPL routines to control access to data

You can use an SPL routine to control access to individual tables and columns in the database. Use a routine to accomplish various degrees of access control. A powerful feature of SPL is the ability to designate an SPL routine as a DBA-privileged routine. When you write a DBA-privileged routine, you can allow users who have few or no table privileges to have DBA privileges when they execute the routine. In the routine, users can carry out specific tasks with their temporary DBA privilege. The DBA-privileged routine lets you accomplish the following tasks:

- You can restrict how much information individual users can read from a table.
- You can restrict all the changes that are made to the database and ensure that entire tables are not emptied or changed accidentally.
- You can monitor an entire class of changes made to a table, such as deletions or insertions.
- You can restrict all object creation (data definition) to occur within an SPL routine so that you have complete control over how tables, indexes, and views are built.

For information about routines in SPL, see the *Informix® Guide to SQL: Tutorial*.

Restrict data reads

The routine in the following example hides the SQL syntax from users, but it requires that users have the Select privilege on the **customer** table. If you want to restrict what users can select, write your routine to work in the following environment:

- You are the DBA of the database.
- The users have the Connect privilege to the database. They do not have the Select privilege on the table.
- You use the DBA keyword to create the SPL routine (or set of SPL routines).
- Your SPL routine (or set of SPL routines) reads from the table for users.

If you want users to read only the name, address, and telephone number of a customer, you can modify the procedure as the following example shows:

```
CREATE DBA PROCEDURE read_customer(cnum INT)
RETURNING CHAR(15), CHAR(15), CHAR(18);

DEFINE p_lname,p_fname CHAR(15);
DEFINE p_phone CHAR(18);

SELECT fname, lname, phone
   INTO p_fname, p_lname, p_phone
   FROM customer
   WHERE customer_num = cnum;

RETURN p_fname, p_lname, p_phone;

END PROCEDURE;
```

Restrict changes to data

When you use SPL routines, you can restrict changes made to a table. Channel all changes through an SPL routine. The SPL routine makes the changes, rather than users making the changes directly. If you want to limit users to deleting one row at a time to ensure that they do not accidentally remove all the rows in the table, set up the database with the following privileges:

- You are the DBA of the database.
- All the users have the Connect privilege to the database. They might have the Resource privilege. They do not have the Delete privilege (for this example) on the table being protected.
- You use the DBA keyword to create the SPL routine.
- Your SPL routine performs the deletion.

Write an SPL procedure similar to the following one, which uses a WHERE clause with the **customer_num** that the user provides, to delete rows from the **customer** table:

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DELETE FROM customer
    WHERE customer_num = cnum;

END PROCEDURE;
```

Monitor changes to data

When you use SPL routines, you can create a record of changes made to a database. You can record changes that a particular user makes, or you can make a record each time a change is made.

You can monitor all the changes a single user makes to the database. Channel all changes through SPL routines that keep track of changes that each user makes. If you want to record each time the user **acctclrk** modifies the database, set up the database with the following privileges:

- You are the DBA of the database.
- All other users have the Connect privilege to the database. They might have the Resource privilege. They do not have the Delete privilege (for this example) on the table being protected.
- You use the DBA keyword to create an SPL routine.
- Your SPL routine performs the deletion and records that a certain user makes a change.

Write an SPL routine similar to the following example (for a UNIX™ platform), which uses a customer number the user provides to update a table. If the user happens to be **acctclrk**, a record of the deletion is put in the file `updates`.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
    WHERE customer_num = cnum;

IF username = 'acctclrk' THEN
    SYSTEM 'echo Delete from customer by acctclrk >>
/mis/records/updates' ;
END IF
END PROCEDURE;
```

To monitor all the deletions made through the procedure, remove the IF statement and make the SYSTEM statement more general. The following procedure changes the previous routine to record all deletions:

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);
LET username = USER ;
DELETE FROM tbname WHERE customer_num = cnum;

SYSTEM
 'echo Deletion made from customer table, by '||username
||'>>/hr/records/deletes';
```

```
END PROCEDURE;
```

Restrict object creation

To put restraints on what objects are built and how they are built, use SPL routines within the following setting:

- You are the DBA of the database.
- All the other users have the Connect privilege to the database. They do not have the Resource privilege.
- You use the DBA keyword to create an SPL routine (or set of SPL routines).
- Your SPL routine (or set of SPL routines) creates tables, indexes, and views in the way you define them. You might use such a routine to set up a training database environment.

Your SPL routine might include the creation of one or more tables and associated indexes, as the following example shows:

```
CREATE DBA PROCEDURE all_objects()

CREATE TABLE learn1 (intone SERIAL, inttwo INT NOT NULL,
   charcol CHAR(10) );
CREATE INDEX learn_ix ON learn1 (inttwo);
CREATE TABLE toys (name CHAR(15) NOT NULL UNIQUE,
   description CHAR(30), on_hand INT);
END PROCEDURE;
```

To use the **all_objects** procedure to control additions of columns to tables, revoke the Resource privilege on the database from all users. When users try to create a table, index, or view with an SQL statement outside your procedure, they cannot do so. When users execute the procedure, they have a temporary DBA privilege so the CREATE TABLE statement, for example, succeeds, and you are guaranteed that every column that is added has a constraint placed on it. In addition, objects that users create are owned by those users. For the **all_objects** procedure, whoever executes the procedure owns the two tables and the index.

Views

A *view* is a synthetic table. You can query it as if it were a table, and in some cases, you can update it as if it were a table. However, it is not a table. It is a synthesis of the data that exists in real tables and other views.

The basis of a view is a SELECT statement. When you create a view, you define a SELECT statement that generates the contents of the view at the time you access the view. A user also queries a view with a SELECT statement. In some cases, the database server merges the select statement of the user with the one defined for the view and then actually performs the combined statements. For information about the performance of views, see your *Informix® Performance Guide*.

Because you write a SELECT statement that determines the contents of the view, you can use views for any of the following purposes:

- To restrict users to particular columns of tables

  You name only permitted columns in the select list in the view.
- To restrict users to particular rows of tables

You specify a WHERE clause that returns only permitted rows.

- To constrain inserted and updated values to certain ranges

You can use the WITH CHECK OPTION (explained on page) to enforce constraints.

- To provide access to derived data without storing redundant data in the database

You write the expressions that derive the data into the select list in the view. Each time you query the view, the data is derived anew. The derived data is always up to date, yet no redundancies are introduced into the data model.

- To hide the details of a complicated SELECT statement

You hide complexities of a multitable join in the view so that neither users nor application programmers must repeat them.

Create views

The following example creates a view based on a table in the **stores_demo** database:

```
CREATE VIEW name_only AS
SELECT customer_num, fname, lname FROM customer
```

The view exposes only three columns of the table. Because it contains no WHERE clause, the view does not restrict the rows that can appear.

The following example is based on the join of two tables:

```
CREATE VIEW full_addr AS
SELECT address1, address2, city, state.sname,
    zipcode, customer_num
    FROM customer, state
    WHERE customer.state = state.code
```

The table of state names reduces the redundancy of the database; it lets you store the full state names only once, which can be useful for long state names such as Minnesota. This **full_addr** view lets users retrieve the address as if the full state name were stored in every row. The following two queries are equivalent:

```
SELECT * FROM full_addr WHERE customer_num = 105

SELECT address1, address2, city, state.sname,
    zipcode, customer_num
    FROM customer, state
    WHERE customer.state = state.code AND customer_num = 105
```

However, be careful when you define views that are based on joins. Such views are not modifiable; that is, you cannot use them with UPDATE, DELETE, or INSERT statements. For an explanation of how to modify with views, see .

The following example restricts the rows that can be seen in the view:

```
CREATE VIEW no_cal_cust AS
    SELECT * FROM customer WHERE NOT state = 'CA'
```

This view exposes all columns of the **customer** table, but only certain rows. The following example is a view that restricts users to rows that are relevant to them:

```
CREATE VIEW my_calls AS
    SELECT * FROM cust_calls WHERE user_id = USER
```

All the columns of the **cust_calls** table are available but only in those rows that contain the user IDs of the users who can execute the query.

Typed views

You can create a typed view when you want to distinguish between two views that display data of the same data type. For example, suppose you want to create two views on the following table:

```
CREATE TABLE emp
(   name    VARCHAR(30),
    age     INTEGER,
    salary  INTEGER);
```

The following statements create two typed views, **name_age** and **name_salary**, on the **emp** table:

```
CREATE ROW TYPE name_age_t
(   name   VARCHAR(20),
    age    INTEGER);

CREATE VIEW name_age OF TYPE name_age_t AS
    SELECT name, age FROM emp;

CREATE ROW TYPE name_salary_t
(   name    VARCHAR(20),
    salary  INTEGER);

CREATE VIEW name_salary OF TYPE name_salary_t AS
    SELECT name, salary FROM emp
```

When you create a typed view, the data that the view displays is of a named row type. For example, the **name_age** and **name_salary** views contain VARCHAR and INTEGER data. Because the views are typed, a query against the **name_age** view returns a column view of type **name_age** whereas a query against the **name_salary** view returns a column view of type **name_salary**. Consequently, the database server is able to distinguish between rows that the **name_age** and **name_salary** views return.

In some cases, a typed view has an advantage over an untyped view. For example, suppose you overload the function **myfunc()** as follows:

```
CREATE FUNCTION myfunc(aa name_age_t) ......;
CREATE FUNCTION myfunc(aa name_salary_t) .....;
```

Because the **name_age** and **name_salary** views are typed views, the following statements resolve to the appropriate **myfunc()** function:

```
SELECT myfunc(name_age) FROM name_age;
SELECT myfunc(name_salary) FROM name_salary;
```

You can also write the preceding SELECT statements using an alias for the table name:

```
SELECT myfunc(p) FROM name_age p;
SELECT myfunc(p) FROM name_salary p;
```

If two views that contain the same data types are not created as typed views, the database server cannot distinguish between the rows that the two views display. For more information about function overloading, see *Informix® User-Defined Routines and Data Types Developer's Guide*.

Duplicate rows from views

A view might produce duplicate rows, even when the underlying table has only unique rows. If the view SELECT statement can return duplicate rows, the view itself can appear to contain duplicate rows.

You can prevent this problem in two ways. One way is to specify DISTINCT in the projection list in the view. However, when you specify DISTINCT, it is impossible to modify with the view. The alternative is to always select a column or group of columns that is constrained to be unique. (You can be sure that only unique rows are returned if you select the columns of a primary key or of a candidate key. Build a relational data model on page 6 contains information about primary and candidate keys.)

Restrictions on views

Because a view is not really a table, it cannot be indexed, and it cannot be the object of such statements as ALTER TABLE and RENAME TABLE. You cannot rename the columns of a view with RENAME COLUMN. To change anything about the definition of a view, you must drop the view and re-create it.

Because it must be merged with the user's query, the SELECT statement on which a view is based cannot contain the following clauses or keywords:

INTO TEMP

The user's query might contain INTO TEMP; if the view also contains it, the data would not know where to go.

ORDER BY

The user's query might contain ORDER BY. If the view also contains it, the choice of columns or sort directions can be in conflict.

A SELECT statement on which you base a view can contain the UNION keyword. In such cases, the database server stores the view in an implicit temporary table where the unions are evaluated as necessary. The user's query uses this temporary table as a base table.

When the basis of the view changes

The tables and views on which you base a view can change in several ways. The view automatically reflects most of the changes.

When you drop a table or view, any views in the same database that depend on that table or view are automatically dropped.

The only way to alter the definition of a view is to drop and re-create it. Therefore, if you change the definition of a view on which other views depend, you must also re-create the other views (because they all are dropped).

When you rename a table, any views in the same database that depend on that table are modified to use the new name. When you rename a column, views in the same database that depend on that table are updated to select the correct column. However, the names of columns in the views themselves are not changed. For an example, recall the following view on the **customer** table:

```
CREATE VIEW name_only AS
    SELECT customer_num, fname, lname FROM customer
```

Now suppose that you change the **customer** table in the following way:

```
RENAME COLUMN customer.lname TO surname
```

To select surnames of customers directly, you must now select the new column name. However, the name of the column as seen through the view is unchanged. The following two queries are equivalent:

```
SELECT fname, surname FROM customer

SELECT fname, lname FROM name_only
```

When you drop a column to alter a table, views are not modified. If views are used, error -217 (`Column not found in any table in the query`) occurs. The reason views are not modified is that you can change the order of columns in a table by dropping a column and then adding a column of the same name. Views based on that table continue to work and they retain their original sequence of columns.

You can base a view on tables and views in external databases. Changes to tables and views in other databases are not reflected in views. Such changes might not be apparent until someone queries the view and gets an error because an external table changed. For example, if a column in a remote object that is included in a view is altered to have a different type, you must drop and re-create the view.

Modify with a view

You can modify views as if they were tables. Some views can be modified and others not, depending on their SELECT statements. The restrictions are different, depending on whether you use DELETE, UPDATE, or INSERT statements.

You can modify a view if the SELECT statement that defined it did not contain any of the following items:

- A join of two or more tables
- An aggregate function or the GROUP BY clause
- The DISTINCT keyword or its synonym, UNIQUE
- The UNION keyword
- Calculated or literal values

When a view avoids all these restricted features, each row of the view corresponds to exactly one row of one table. By using INSTEAD OF triggers, you can circumvent these restrictions on the view if the trigger action modifies the base table.

Delete with a view

You can use a DELETE statement on a modifiable view as if it were a table. The database server deletes the correct row of the underlying table.

Update a view

You can use an UPDATE statement on a modifiable view. However, the database server does not support updating any derived column. A *derived* column is a column produced by an expression in the select list of the CREATE VIEW statement (for example, `order_date + 30`).

The following example shows a modifiable view that contains a derived column and an UPDATE statement that can be accepted against it:

```
CREATE    VIEW   response(user_id, received, resolved, duration) AS
   SELECT user_id, call_dtime, res_dtime,  res_dtime - call_dtime
      FROM cust_calls
      WHERE user_id = USER;

UPDATE response SET resolved = TODAY
   WHERE resolved IS NULL;
```

You cannot update the duration column of the view because it represents an expression (the database server cannot, even in principle, decide how to distribute an update value between the two columns that the expression names). But if no derived columns are named in the SET clause, you can perform the update as if the view were a table.

A view can return duplicate rows even though the rows of the underlying table are unique. You cannot distinguish one duplicate row from another. If you update one of a set of duplicate rows (for example, if you use a cursor to update WHERE CURRENT), you cannot be sure which row in the underlying table receives the update.

Insert into a view

You can insert rows into a view only if the view is modifiable and contains no derived columns. The reason for the second restriction is that an inserted row must provide values for all columns, but the database server cannot tell how to distribute an inserted value through an expression. An attempt to insert into the **response** view, as the previous example shows, would fail.

When a modifiable view contains no derived columns, you can insert into it as if it were a table. The database server, however, uses NULL as the value for any column that is not exposed by the view. If such a column does not allow NULL values, an error occurs, and the insert fails.

Another mechanism for inserting rows (or performing UPDATE or DELETE operations) on views, including complex views, is to create INSTEAD OF triggers, as described in the *Informix® Guide to SQL: Syntax*.

Use the WITH CHECK OPTION keywords

You can insert into a view a row that does not satisfy the conditions of the view; that is, a row that is not visible through the view. You can also update a row of a view so that it no longer satisfies the conditions of the view.

To avoid updating a row of a view so that it no longer satisfies the conditions of the view, add the WITH CHECK OPTION keywords when you create the view. This clause asks the database server to test every inserted or updated row to ensure that it meets the conditions set by the WHERE clause of the view. The database server rejects the operation with an error if the conditions are not met.

🚫 **Restriction:** You cannot include the WITH CHECK OPTION keywords when a UNION operator is included in the view definition.

In the previous example, the view named **response** is defined as the following example shows:

```
CREATE   VIEW   response (user_id, received,  resolved,  duration) AS
   SELECT user_id,call_dtime,res_dtime,res_dtime  - call_dtime
      FROM cust_calls
      WHERE user_id = USER
```

You can update the **user_id** column of the view, as the following example shows:

```
UPDATE response SET user_id = 'lenora'
   WHERE received BETWEEN TODAY AND TODAY – 7
```

The view requires rows in which **user_id** equals USER. If user **tony** performs this update, the updated rows vanish from the view. You can create the view, however, as the following example shows:

```
CREATE   VIEW    response  (user_id, received, resolved,duration) AS
   SELECT user_id, call_dtime, res_dtime, res_dtime – call_dtime
      FROM cust_calls
      WHERE user_id = USER
WITH CHECK OPTION
```

The preceding UPDATE operation by user **tony** is rejected as an error.

You can use the WITH CHECK OPTION feature to enforce any kind of data constraint that can be stated as a Boolean expression. In the following example, you can create a view of a table for which you express all the logical constraints on data as conditions of the WHERE clause. Then you can require all modifications to the table to be made through the view.

```
CREATE VIEW order_insert AS
   SELECT * FROM orders O
      WHERE order_date = TODAY -- no back-dated entries
         AND EXISTS -- ensure valid foreign key
            (SELECT * FROM customer C
               WHERE O.customer_num = C.customer_num)
         AND ship_weight < 1000 -- reasonableness checks
         AND ship_charge < 1000
WITH CHECK OPTION
```

Because of EXISTS and other tests, which are expected to be successful when the database server retrieves existing rows, this view displays data from **orders** inefficiently. However, if insertions to **orders** are made only through this view (and you do not already use integrity constraints to constrain data), users cannot insert a back-dated order, an invalid customer number, or an excessive shipping weight and shipping charge.

Re-execution of a prepared statement when the view definition changes

The database server uses the definition of the view that exists when you prepare a SELECT statement with that view. If the definition of a view changes after you prepare a SELECT statement on that view, the execution of the prepared statement gives incorrect results because it does not reflect the new view definition. No SQL error is generated.

Privileges and views

When you *create* a view, the database server tests your discretionary access privileges on the underlying tables and views. When you *use* a view, however, only your privileges on that view are tested.

After access privileges on tables referenced in view definitions are modified, the database server does not automatically apply to existing views whatever changes were made to access privileges on the underlying tables. To force modified privileges on any table from which the view is derived to apply also to the view, use the DROP VIEW and CREATE VIEW statements of SQL to drop and recreate the view.

Dropping a view also destroys any other views and INSTEAD OF triggers whose definitions depend on that view. After you drop and recreate a view to synchronize its access privileges with those of its underlying tables, you can use the CREATE TRIGGER and CREATE VIEW statements to recreate, respectively, any INSTEAD OF triggers and dependent views that the database server destroyed when you dropped the view.

Privileges when creating a view

The database server tests to make sure that you have all the privileges that are required to execute the SELECT statement in the view definition. If you do not, the database server does not create the view.

This test ensures that users cannot create a view on the table and query the view to gain unauthorized access to a table.

After you create the view, the database server grants you, the creator and owner of the view, at least the Select privilege on it. No automatic grant is made to PUBLIC, as is the case with a newly created table.

The database server tests the view definition to see if the view is modifiable. If it is, the database server grants you the Insert, Delete, and Update privileges on the view, provided that you also have those privileges on the underlying table or view. In other words, if the new view is modifiable, the database server copies your Insert, Delete, and Update privileges from the underlying table or view and grants them on the new view. If you have only the Insert privilege on the underlying table, you receive only the Insert privilege on the view.

This test ensures that users cannot use a view to gain access to any privileges that they did not already have.

Because you cannot alter or index a view, the Alter and Index privileges are never granted on a view.

This section does not apply to views on remote tables. Permissions on remote tables are not propagated automatically to views on those tables. To provide PUBLIC with Select access to a view that includes one or more columns in a remote table,

for example, you must explicitly execute REVOKE ALL FROM PUBLIC for the view, and then explicitly grant Select privilege on that view to PUBLIC.

Privileges when using a view

When you attempt to use a view, the database server tests only the privileges that you are granted on the view. It does not test your right to access the underlying tables.

If you create the view, your privileges are the ones noted in the preceding section. If you are not the creator, you have the privileges that the creator (or someone who had the WITH GRANT OPTION privilege) granted you.

Therefore, you can create a table and revoke access of PUBLIC to it; then you can grant limited access privileges to the table through views. Suppose you want to grant access privileges on the following table:

```
CREATE TABLE hr_data
   (
   emp_key INTEGER,
   emp_name CHAR(40),
   hire_date DATE,
   dept_num SMALLINT,
   user-id CHAR(18),
   salary DECIMAL(8,2),
   performance_level CHAR(1),
   performance_notes TEXT
   )
```

The section Column-level privileges on page 66 shows how to grant access privileges directly on the **hr_data** table. The examples that follow take a different approach. Assume that when the table was created, this statement was executed:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

(Such a statement is not necessary in an ANSI-compliant database.) Now you create a series of views for different classes of users. For users who should have read-only access to the nonsensitive columns, you create the following view:

```
CREATE VIEW hr_public AS
   SELECT emp_key, emp_name, hire_date, dept_num, user_id
      FROM hr_data
```

Users who are given the Select privilege for this view can see nonsensitive data and update nothing. For Human Resources personnel who must enter new rows, you create a different view, as the following example shows:

```
CREATE VIEW hr_enter AS
   SELECT emp_key, emp_name, hire_date, dept_num
      FROM hr_data
```

You grant these users both Select and Insert privileges on this view. Because you, the creator of both the table and the view, have the Insert privilege on the table and the view, you can grant the Insert privilege on the view to others who have no privileges on the table.

On behalf of the person in the MIS department who enters or updates new user IDs, you create still another view, as the following example shows:

```
CREATE VIEW hr_MIS AS
   SELECT emp_key, emp_name, user_id
      FROM hr_data
```

This view differs from the previous view in that it does not expose the department number and date of hire.

Finally, the managers require access to all columns and they require the ability to update the performance-review data for their own employees only. You can meet these requirements by creating a table, **hr_data**, that contains a department number and computer user IDs for each employee. Let it be a rule that the managers are members of the departments that they manage. Then the following view restricts managers to rows that reflect only their employees:

```
CREATE VIEW hr_mgr_data AS
   SELECT * FROM hr_data
      WHERE dept_num =
         (SELECT dept_num FROM hr_data
             WHERE user_id = USER)
      AND NOT user_id = USER
```

The final condition is required so that the managers do not have update access to their own row of the table. Therefore, you can safely grant the Update privilege to managers for this view, but only on selected columns, as the following statement shows:

```
GRANT SELECT, UPDATE (performance_level, performance_notes)
   ON hr_mgr_data TO peter_m
```

Distributed queries

This chapter provides an overview of distributed queries. Distributed queries allow shared access to data across multiple databases within a network of Informix® database servers. Different database servers can manage multiple databases, which can be referenced in a single distributed query.

Overview of distributed queries

The Informix® database servers allows you to query more than one database of the same database server or across multiple database servers. This type of query is called a *distributed query*. The database servers can be located on a single host computer, on different computers of the same network, or on a gateway. (In general, most features and restrictions that this chapter describes for distributed queries also apply to function calls and to distributed INSERT, DELETE, or UPDATE operations that reference objects or data in more than one database.)

Distributed queries across databases of one Informix® instance

Cross-database queries that access tables in more than one database of the local server instance can return built in-atomic data types that are not opaque, and most built-in opaque data types, and user-defined atomic types (UDTs) that have the same definitions in all participating databases.

Data values returned from distributed operations across databases of the same HCL Informix® server instance are restricted, however, to the following returned data types:

- The *built-in atomic data types* that are not opaque, including these:
    ◦ BIGINT
    ◦ BIGSERIAL
    ◦ BYTE
    ◦ CHAR

- ◦ DATE
- ◦ DATETIME
- ◦ DECIMAL
- ◦ FLOAT
- ◦ INT
- ◦ INTERVAL
- ◦ INT8
- ◦ MONEY
- ◦ NCHAR
- ◦ NVARCHAR
- ◦ SERIAL
- ◦ SERIAL8
- ◦ SMALLFLOAT
- ◦ SMALLINT
- ◦ TEXT
- ◦ VARCHAR
- Most *built-in opaque data types*, including these:
  - ◦ BLOB
  - ◦ BOOLEAN
  - ◦ BSON
  - ◦ CLIENTBINVAL
  - ◦ CLOB
  - ◦ IFX_LO_SPEC
  - ◦ IFX_LO_STAT
  - ◦ INDEXKEYARRAY
  - ◦ JSON
  - ◦ LVARCHAR
  - ◦ POINTER
  - ◦ RTNPARAMTYPES,
  - ◦ SELFUNCARGS
  - ◦ STAT
  - ◦ XID
- For user-defined data types (UDTs), the query, DML operation, or function call can return DISTINCT or OPAQUE data types that are explicitly cast to a built-in atomic data type. Each of these DISTINCT and OPAQUE data types, and all of their explicit casts, must be defined identically in each participating database that stores or receives those data values.

Data types unsupported in cross-database queries

The cross-database distributed query fails, however, if it references a table, view, or synonym in another database of the local HCL Informix® instance that includes a column of any of the following built-in opaque or complex data types:

- LOLIST
- IMPEXP
- IMPEXPBIN
- SENDRECV
- DISTINCT of any of these built-in opaque data types
- DISTINCT of any of the DISTINCT opaque data types immediately above
- Complex types, including COLLECTION, LIST, MULTISET, SET, and named or unnamed ROW.

Distributed queries across databases of two or more Informix® instances

Distributed queries that access tables in databases of two or more participating server instances can return built in-atomic data types that are not opaque, and can return some atomic built-in opaque data types. Cross-server queries can also return values of user-defined DISTINCT types of built-in atomic types, if those DISTINCT values are explicitly cast to built-in types, and if all participating databases use the same DISTINCT definitions and cast definitions.

Distributed operations across databases of two or more HCL Informix® instances are subject to the following restrictions on returned data types:

- A cross-server query, DML operation, or function call can return non-opaque atomic built-in data types, and the built-in opaque BOOLEAN data type, the BSON and JSON key-value pair types, and the LVARCHAR data type.
- A cross-server query, DML operation, or function call can return DISTINCT data types that are explicitly cast to a built-in atomic data type, and whose base types are either non-opaque built-in data types, or BOOLEAN or LVARCHAR, or BSON or JSON key-value pair types.
- Additionally, the base type can also be a DISTINCT data type whose base type is a non-opaque built-in type, or a BOOLEAN, LVARCHAR, or BSON or JSON key-value pair types, or whose base type is another DISTINCT data type that is based on one of these types.
- Each participating database of the distributed operation must define these explicit casts, functions, and DISTINCT data types identically.
- If any participating database servers are earlier versions that cannot support some data types in cross-server operations, those servers return only the data types that they support. A distributed operation fails if that operation specifies an unsupported data type.

Like distributed operations across databases of the same Informix® server instance, cross-server distributed operations require that all databases be of compatible transaction logging types, as described in .

Data types that cross-server queries cannot return

The following built-in data types are not supported in distributed cross-server queries. The cross-server query (or any other cross-server DML operation) fails with an error if it references a table, view, or synonym in a database of another Informix® instance that includes a column of any of the following large-object, opaque, or complex data types:

- BLOB
- CLOB
- BYTE

- TEXT
- CLIENTBINVAL
- IFX_LO_SPEC
- IFX_LO_STAT
- IMPEXP
- IMPEXPBIN
- INDEXKEYARRAY
- LOLIST
- POINTER
- RTNPARAMTYPES
- SELFUNCARGS
- SENDRECV
- STAT
- STREAM
- XID
- User-defined OPAQUE types
- Complex types (COLLECTION, LIST, MULTISET, SET, and named or unnamed ROW)
- DISTINCT of any data type that is listed above.

Coordinator and participant servers in a distributed query

Queries that access more than one database server are called *cross-server* queries. To support distributed DML operations across multiple database servers, Informix® servers maintain hierarchical relationships between a coordinator server and one or more participant servers.

Here *coordinator* and *participant* are defined as follows:

- The coordinator directs the resolution of the query. It also decides whether the query should be committed or cancelled.
- The participant (also called the *subordinate* server) directs the execution of the distributed query on one *branch*. The branch is the part of the distributed query involving only that participant database server.

More generally, the server of the local database of a session that initiates any cross-server distributed DML operation is called the *coordinator*, and the subordinate servers that execute individual branches of the same distributed transaction are called *participant* servers. The term *distributed query* is sometimes applied to any cross-server DML operation, including DELETE, INSERT, MERGE and UPDATE statements.

Cross-server distributed operations can include multiple database server instances as subordinate participants but exactly one coordinator. The coordinator must establish a connection with each subordinate. Within a single distributed operation, however, if a subordinate server attempts to connect to another subordinate, or to establish a new connection with the coordinator, the distributed operation fails with an error.

The distributed query example that follows refers to objects in a multi-server environment,

- where db is the local database,
- db2 is another database located on the same server,
- and master_db is an external database on the remote server new_york.

The following example shows a cross-server distributed query that accesses data on remote server new_york, using the local server of database db as the transaction coordinator.

```
DATABASE db;
SELECT col2 FROM db2:tab1, master_db@newyork:tab2;
```

At a given time, a session can have only one local database, but can open multiple external databases. Distributed queries must always originate on the database server instance that acts as the coordinator.

Calling remote routines in cross-server operations

This restriction of the topology of cross-server connections to exactly two levels, the coordinator and the participants, limits the calling context for remote UDRs within cross-server operations to the coordinator.

For example, in the multi-server environment of the previous example, suppose that your local database included a UDR called bad_example1() that the following CREATE PROCEDURE statement had defined:

```
CREATE PROCEDURE bad_example1()
    EXECUTE PROCEDURE master_db@new_york:bad_example2()
END PROCEDURE;
```

Suppose also that the remote UDR called bad_example2() in the master_db database of the new_york database server instance had the following definition that invokes a remote UDR called bad_example3() on the new_buffalo database server instance:

```
CREATE PROCEDURE bad_example2()
    EXECUTE PROCEDURE examples_db@new_buffalo:bad_example3()
END PROCEDURE;
```

Suppose also that the remote UDR bad_example3() exists in the examples_db database of the new_buffalo database server, and that you hold all the required access privileges to invoke the local instance, the new_york instance, and the new_buffalo instance of the bad_example3() routine.

In this environment, when you issue the following statement from the local database:

```
EXECUTE PROCEDURE bad_example1();
```

your local database server, as the coordinator of this cross-server distributed operation, invokes the remote UDR bad_example2() on the new_york database server instance. Error `-556` is returned, however, when the remote bad_example2() routine attempts to establish a connection between the new_york and the new_buffalo database servers. That attempt to connect fails, because it violates the rule that only the coordinator can connect to participant servers in the distributed transaction.

An invocation of the same remote UDR would have succeeded, however, if your session had not invoked bad_example1(), but a session on a local database of the new_york database server instance had instead initiated the cross-server distributed operation by issuing the

```
EXECUTE PROCEDURE bad_example2();
```

statement. In this case, the new_york database server is the coordinator of a cross-server operation in which the new_buffalo database server is a subordinate participant on the branch of the distributed operation that executes bad_example3(), and there is no third level of cross-server connections.

Remember:

Cross-server operations require exactly one coordinator to connect to one or more database server instances as subordinate participants. If a participant attempts to access any remote object, or to call a remote UDR in the database of any other server instance, that invocation fails with error -556, because only the coordinator can establish a connection with another database server.

Configure the database server to use distributed queries

To use multiple Informix® servers for distributed queries, you must make sure that all of the database servers involved are configured to enable server-to-server communications over the network.

Edit the following configuration files to allow distributed queries:

- The `sqlhosts` file to hold connectivity information about other servers
- The `onconfig` file to set DBSERVERALIASES, NETTYPE, REMOTE_USERS_CFG, and REMOTE_SERVER_CFG configuration parameters
- The file specified by the REMOTE_USERS_CFG or REMOTE_SERVER_CFG configuration parameter to configure network security
- The `/etc/services` and `/etc/hosts` files, or their equivalents managed via network systems such as NIS+, for TCP/IP network configuration

Note: To configure network security, use the file you specify with the REMOTE_USERS_CFG or REMOTE_SERVER_CFG configuration parameter instead of the `hosts.equiv` or trusted users' `rhosts` files.

To set up several database servers to use distributed queries, use one of the following ways to store the `sqlhosts` information for all the databases:

- In one `sqlhosts` file, pointed to by the **INFORMIXSQLHOSTS** environment variable
- In separate `sqlhosts` files in each database server directory
- In a centrally managed file on a network mounted, read-only file system, with the sqlhosts file in each database server directory being a symbolic link to the centrally managed file

Note: To use non-root installations of the database server for distributed queries, you must set one of the following configuration parameters in the `onconfig` file:

> • REMOTE_USERS_CFG, which specifies the alternative to using the `rhosts` file for listing trusted users on a remote server.
> • REMOTE_SERVER_CFG, which specifies the alternative to using the `etc/hosts.equiv` file for listing trusted remote hosts.

Syntax of a distributed query

This section describes how to specify a remote server, database, and database object within a distributed query.

Access a remote server and database

The core element of any statement within a distributed query is the database segment. Using the syntax of both of these segments, you can specify a remote database server, database, or database object.

Database Name segment

The Database Name segment is used to specify the name of a database. The following examples show different ways of specifying a remote database:

```
empinfo@personnel '//personnel/empinfo'
```

Database Object Name segment

The Database Object Name segment is used to specify the name of a database object, including constraints, indexes, triggers, any synonyms. The following examples show how to access remote objects:

```
empinfo@personnel:markg.emp_names empinfo@personnel:emp_names
```

Valid statements for accessing remote objects

The following statements support remote objects as part of the Database and Database Object segments and can be used within a distributed query:

- INSERT
- SELECT
- UPDATE
- DELETE
- CREATE VIEW
- CREATE SYNONYM
- CREATE DATABASE
- DATABASE
- LOAD
- UNLOAD
- LOCK

- UNLOCK
- INFO

Access remote tables

A remote table is a table on a database server other than the current server. You can connect from your current server to a remote server.

At any time, there can be only one active connection from the local server to a remote server. Informix® does not support multiple active connections between the same two database servers using different server aliases. Thus, if you use different server aliases to connect to the same remote server, the initial connection is reused.

The general syntax for accessing a table on another server is:

```
database@server:[owner.]table
```

Here, a table can be a table name, view name or synonym. You have the option of specifying the table owner. For the complete syntax options, see the documentation of the Database and Database Object segments in the *Informix® Guide to SQL: Syntax*.

The following example shows a query that accesses a remote table:

```
DATABASE locdb; SELECT l.name, r.assignment FROM rdb@rsys:rtab r,
loctab l WHERE l.empid = r.empid;
```

This query accesses the name and empid columns from the local table loctab, and the assignment and empid columns from the remote table rtab. The data is joined using empid as the join column.

The following example shows a query that accesses data on a remote table and inserts it into a local table:

```
DATABASE locdb; INSERT INTO loctab SELECT * FROM rdb@rsys:rtab;
```

This query selects all data from the remote table rtab, and inserts it into the local table loctab.

The following example creates a view in the local database using the empid and priority columns from the remote database rdb.

```
DATABASE locdb; CREATE VIEW myview (empid, empprty)
AS SELECT empid, priority FROM rdb@rsys:rtab;
```

Table permissions

Permissions for accessing table in other databases and remote tables are controlled at the table location. When accessing a remote server, the connection is made using the login name and password of the user executing the query. To access remote data, the user must have the correct permissions on the remote table.

When processing distributed queries, the database server ignores the active role on the current local database when accessing a remote object. On the remote server, the default role applied to each remote database is used. If a default role is not defined, the user's privilege define the access permissions for the objects in each remote database.

Qualify table references

References to tables may be qualified with the current database and server name. If no qualification is specified, the current database and server context is implied. For example, if the current database is locdb and the current server is currsys, the following references to loctab are equivalent:

```
locdb@currsys:loctab
locdb:loctab
loctab
```

Other remote operations

In addition to querying and updating data, there are other remote operations that you can perform using the distributed query framework.

Open a remote database

By specifying a remote object in the DATABASE statement, you can open a remote database as in the following examples:

```
DATABASE dbname@servername;
DATABASE "//servernam/database";
```

Create a remote database

You can create a remote database by qualifying the database name with a server name when using the CREATE DATABASE statement.

```
CREATE DATABASE remfoo@rsys;
```

Create a synonym for a remote table

You can create a synonym for a remote table in another database or a remote table using a qualified name in the CREATE SYNONYM statement. For example, the following statement creates a synonym for **rdb@srsys:rtab:**

```
CREATE SYNONYM myrtab FOR rdb@rsys:rtab;
```

It is possible for a synonym to exist in both the local and remote server. In the previous example, it is possible that rtab is itself a synonym for rdb2@rsys2:rtab2. The chain of synonyms is followed when retrieving catalog information until the physical database and server where the table is located are found. If a synonym ultimately points back to itself, an error is returned.

Monitor distributed queries

Use the onstat -x command to display transaction information originating on the coordinator of a distributed query.

The following flag codes in position 5 are used for distributed queries:

C

   Distributed query coordinator

S

> Distributed query participant

B

> Both distributed query coordinator and participant

For more information about using onstat -x see your .

Server environment and distributed queries

Set the DEADLOCK_TIMEOUT configuration parameter and the PDQPRIORITY environment variable to specify information for distributed queries.

The DEADLOCK_TIMEOUT configuration parameter specifies the maximum number of seconds that a database server thread can wait to acquire a lock. If a distributed transaction is forced to wait longer than the number of seconds specified, the thread that owns the transaction assumes that a multi-server deadlock exists. The following error message is returned:

```
-143 ISAM error: deadlock detected.
```

The effective value of **PDQPRIORITY** for a session is sent to the remote site when a connection is established. Subsequent changes to this parameter in the coordinator are not reflected on the remote site. However, the exact behavior of this environment variable depends on the role of the database server in the distributed query (coordinator or participant).

PDQPRIORITY has different syntax and semantics for different server versions. For information about setting **PDQPRIORITY**, see the *Informix® Performance Guide* for your server.

Logging-type restrictions on distributed queries

To execute distributed queries in the Informix® database server environment, all participating databases must be of compatible transaction-logging types:

- Distributed queries are supported on an ANSI-compliant database only if all of the participating databases are also ANSI-compliant.
- Distributed queries on a database that does not support transaction logging are supported only if all of the participating databases also do not use transaction logging.
- Distributed queries on a database that is not ANSI-compliant but that uses explicit transaction logging are supported if all of the other databases also use explicit transaction logging.

In the last case, whether a participating database uses buffered or unbuffered logging does not affect its ability to support distributed operations. In the X/Open distributed transaction processing (DTP) environment, all databases must use unbuffered logging. See the *Informix® Administrator's Guide* for more information about database logging types and X/Open DTP.

Transaction processing

When using distributed queries in a transaction processing environment, be aware of the effect of the isolation level of a transaction, the effect of the SET LOCK MODE statement in conjunction with the DEADLOCK_TIMEOUT configuration parameter, and the two-phase commit protocol.

Isolation levels

The isolation level of a transaction is sent to the remote server at the start of the transaction at the remote site. If an isolation level changes during a transaction, the new value is sent to the remote site.

Related information

DEADLOCK_TIMEOUT and SET LOCK MODE

When working with distributed queries, you can use the SET LOCK MODE statement in conjunction with the **DEADLOCK_TIMEOUT** configuration parameter to help prevent server deadlock.

When you request the WAIT option of SET LOCK MODE, the database server protects against the possibility of a deadlock. However, if the database server discovers that a deadlock can occur, it terminates the operation and returns an error.

The **DEADLOCK_TIMEOUT** configuration parameter specifies the maximum number of seconds that a database server thread can wait to acquire a lock. This value is the default value used by the SET LOCK MODE WAIT statement. This value applies only if you acquire locks on the current and remote database server within the same transaction.

Related information

Two-phase commit and recovery

The two-phase commit protocol is used to ensure that distributed queries are uniformly committed or rolled back across multiple database servers. A database server automatically uses the two-phase commit protocol for any transaction that modifies data on multiple database servers.

Related information

Object-relational databases

Create and use extended data types in Informix®

This chapter describes extended data types that you can use to build an object-relational database. The term *object-relational* is not associated with a particular method or model of database design, but instead refers to any database that uses HCL Informix® features to extend the functionality of the database.

An object-relational database is not antithetical to a relational database but rather is an extension of functionality already present in a relational database. Typically, you use some combination of features from Informix® to extend the kinds of data that your database can store and manipulate. These features include extended data types, smart large objects, type and table inheritance, user-defined casts, and user-defined routines (UDRs). The chapters in this section of the manual describe many of these features. For information about UDRs, see *Informix® User-Defined Routines and Data Types Developer's Guide* and the *Informix® Guide to SQL: Tutorial*.

For an example of an object-relational database, you can create the **superstores_demo** database, which contains examples of some of the features available with Informix®. For information about how to create the **superstores_demo** database, see the *Informix® DB-Access User's Guide*.

HCL Informix® data types

provides a chart for selecting correct data types for the columns of a table depending on the type of data that will be stored. The following figure shows a hierarchy of data types that reflects how the database server manages the data types.

Figure 15. Informix® data types
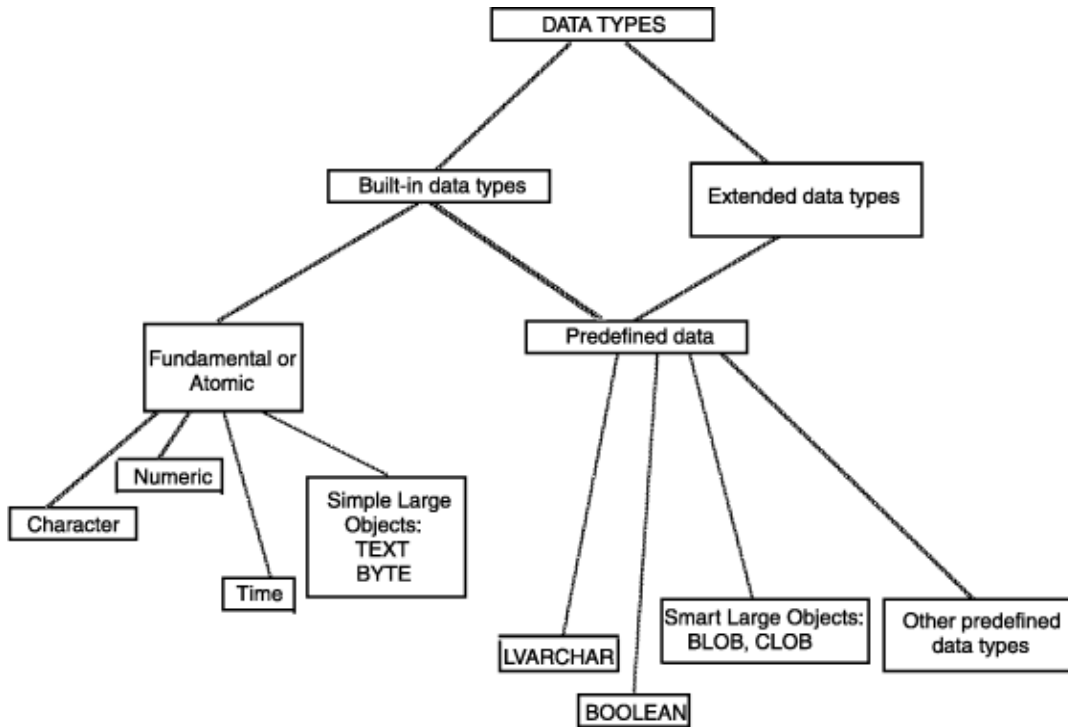
Fundamental or atomic data types

All Informix® database servers support the *fundamental*, or *atomic*, data types. These types are fundamental because they are the smallest units that you can specify in a SELECT statement. Only HCL Informix® supports extended and predefined data types. The predefined data types are in a separate category because they share certain characteristics with extended data types but are provided by the database server.

For an explanation of the fundamental data types, see Select data types on page 21.

Predefined data types

The database server provides the predefined data types, just as it provides the fundamental data types. However, the predefined data types have certain characteristics in common with the extended data types.

BOOLEAN and LVARCHAR data types

BOOLEAN and LVARCHAR data types behave like built-in data types except that the system catalog tables define them as extended data types.

For more information, see Select data types on page 21 and to the system catalog tables in the *Informix® Guide to SQL: Reference*.

IDSSECURITYLABEL data type

The IDSSECURITYLABEL data type stores a security label in a table that is protected by a security policy. Only a user who holds the DBSECADM role can create, alter, or drop a column of this data type. This is a built-in

DISTINCT OF VARCHAR(128) data type, but it is not classified as a character data type because its use is restricted to label-based access control (LBAC).

For more information, see the system catalog tables in the *Informix® Guide to SQL: Reference* and to *Informix® Security Guide*.

BLOB and CLOB data types

The BLOB and CLOB data types are not fundamental data types because you can randomly access data from within the BLOB or CLOB. You can create a table with BLOB and CLOB columns, but you cannot insert data directly into the column. You must use functions to insert and manipulate the data.

For more information, see Smart large objects on page 101.

BSON and JSON data types

The BSON and JSON data types are built-in opaque data types that emulate the functionality of field-value pair data types in NoSQL document stores. On tables with BSON or JSON columns, if you define multiple indexes, with each index based on a single field, you can use those indexes to access subsets of the documents in BSON or JSON collections. By calling built-in functions, queries can retrieve rows whose BSON and JSON column values correspond to specific SQL data types.

Other predefined data types

With the exception of BLOB, BOOLEAN, BSON, CLOB, JSON, and LVARCHAR, the predefined data types usually do not appear as data types for the columns of a table. Instead, the following predefined data types are used with the functions associated with complex and user-defined data types and user-defined routines:

- clientbinval
- ifx_lo_spec
- ifx_lo_stat
- impexp
- impexpbin
- indexkeyarray
- lolist
- pointer
- rtnparamtypes
- selfuncargs
- sendrecv
- stat
- stream
- xid

For more information about these predefined data types, see *Informix® User-Defined Routines and Data Types Developer's Guide*.

Extended data types

Extended data types let you create data types to characterize data that cannot be easily represented with the built-in data types. However, you cannot use extended data types in distributed transactions. The following figure shows the extended data types.

Figure 16. Extended data types
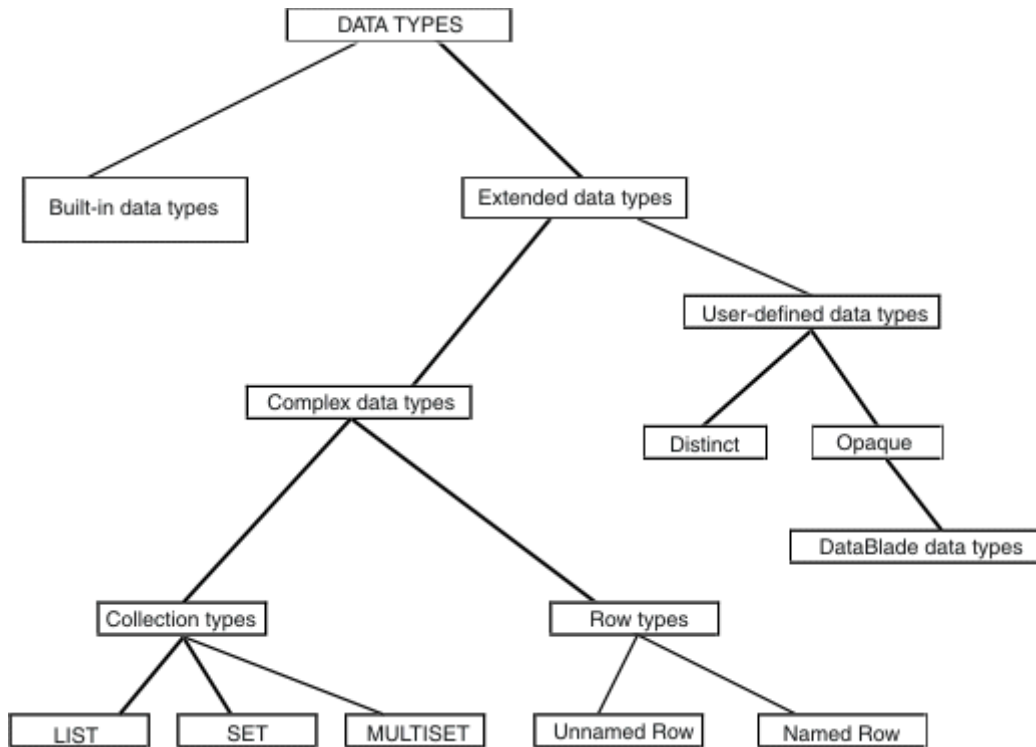
Complex data types

Complex data types describe either a collection of data objects, all of one type (LIST, SET, and MULTISET), or groups of objects of different types (named and unnamed rows.)

User-defined data types

A *user-defined data type* (UDT) is a data type that is not provided by the database server. You must provide all of the information that the database server requires to manage opaque data types or distinct data types.

Informix® supports two categories of user-defined data types, distinct data types and opaque data types.

Distinct data types

A *distinct data type* is an encapsulated data type that you create with the CREATE DISTINCT TYPE statement. A distinct data type has the same representation as, but is distinct from, the data type on which it is based. You can create a distinct data type from built-in types, opaque types, named row types, or other distinct types. You cannot create a distinct data type from any of the following data types:

- BIGSERIAL, SERIAL, and SERIAL8
- Collection types
- Unnamed row types

When you create a distinct data type, you implicitly define the structure of the data type because a distinct data type inherits the structure of its source data type. You can also define functions, operators, and aggregates that operate on the distinct data type.

For information about distinct data types, see Cast distinct data types on page 136, the *Informix® Guide to SQL: Syntax*, and the *Informix® Guide to SQL: Reference*.

Opaque data types

An *opaque data type* is an encapsulated data type that you create with the CREATE OPAQUE TYPE statement. When you create an opaque data type, you must explicitly define the structure of the data type and the functions, operators, and aggregates that operate on the opaque data type. You can use an opaque data type to define columns and program variables in the same way that you use built-in types.

For information about creating opaque data types, see *Informix® User-Defined Routines and Data Types Developer's Guide* and the *Informix® Guide to SQL: Syntax*.

DataBlade® data types

The diagram in Figure 16: Extended data types on page 100 includes DataBlade® data types. A DataBlade® is a suite of user-defined data types and user-defined routines that provides tools for a specialized application. For example, different DataBlade® data types provide tools for managing images, video, and geographical information. Such applications often require opaque data types and other user-defined data types. For information about developing a DataBlade® module, see the *Informix® DataBlade® API Programmer's Guide*.For information about the DataBlade® modules that provides, contact your customer representative.

Smart large objects

Smart large objects are objects that are defined on a BLOB or CLOB data type. A smart large object allows an application program to randomly access column data, which means that you can read or write to any part of a BLOB or CLOB column in any arbitrary order. You can create BLOB or CLOB columns to store binary data or character data.

BLOB data type

You can use a BLOB data type to store any data that a program can generate: graphic images, satellite images, video clips, audio clips, or formatted documents saved by any word processor or spreadsheet. The database server permits any kind of data of any length in a BLOB column.

Like CLOB objects, BLOB objects are stored in whole disk pages in separate disk areas from normal row data.

The advantage of the BLOB data type, as opposed to CLOB, is that it accepts any data. Otherwise, the advantages and disadvantages of the BLOB data type are the same as for the CLOB data type.

CLOB data type

You can use the CLOB data type to store a block of text. It is designed to store ASCII text data, including formatted text such as HTML or PostScript™. Although you can store any data in a CLOB object, Informix® tools expect a CLOB object to be printable, so restrict this data type to printable ASCII text.

CLOB values are not stored with the rows of which they are a part. They are allocated in whole disk pages, usually areas away from rows. (For more information, see your *Informix® Administrator's Guide*.)

The CLOB data type is similar to the TEXT data type except that the CLOB data type provides the following advantages:

- An application program can read from or write to any portion of the CLOB object.
- Access times can be significantly faster because an application program can access any portion of a CLOB object.
- Default characteristics are relatively easy to override. Database administrators can override default characteristics for sbspace at the column level. Application programmers can override some default characteristics for the column when they create a CLOB object.
- You can use the equals operator (=) to test whether two CLOB values are equal.
- A CLOB object is recoverable in the event of a system failure and obeys transaction isolation modes when the DBA or application programmer specifies it. (Recovery of CLOB objects requires that your database system has the necessary resources to provide buffers large enough to handle CLOB objects.)
- You can use the CLOB data type to provide large storage for a user-defined data type.
- DataBlade® developers can create indexes on CLOB data types.

The disadvantages of the CLOB data type are as follows:

- It is allocated in whole disk pages, so a short item wastes space.
- Restrictions apply on how you can use a CLOB column in an SQL statement. (See Use smart large objects on page 102.)
- It is unavailable with all Informix® database servers.

Use smart large objects

To store columns of a BLOB or CLOB data type, you must allocate an *sbspace*. An sbspace is a logical storage unit that stores BLOB and CLOB data in the most efficient way possible. You can write Informix® ESQL/C programs that allow users to fetch and store BLOB or CLOB data. Application programmers who want to access and manipulate smart large objects directly can consult the *Informix® ESQL/C Programmer's Manual*.

In any SQL statement, interactive or programmed, a BLOB or CLOB column cannot be used in the following contexts:

- In arithmetic or Boolean expressions
- In a GROUP BY or ORDER BY clause
- In a LIKE or MATCHES condition
- In a UNIQUE test
- For indexing, as part of a B-tree index

    DataBlade® developers, however, can create indexes on CLOB columns.

In a SELECT statement entered interactively, a BLOB or CLOB column can:

- Specify NULL values as a default when you create a table with the DEFAULT NULL clause
- Disallow NULL values using the NOT NULL constraint when you create a table
- Be tested with the IS [NOT] NULL predicate

From an ESQL/C program, you can use the ifx_lo_stat() function to determine the length of BLOB or CLOB data.

Copy smart large objects

provides functions that you can call from within an SQL statement to import and export smart large objects. The following table shows the smart-large-object functions.

Table 3. SQL functions for smart large objects

Function Name	Purpose
FILETOBLOB()	Copies a file into a **BLOB** column
FILETOCLOB()	Copies a file into a **CLOB** column
LOCOPY()	Copies BLOB or CLOB data into another **BLOB** or **CLOB** column
LOTOFILE()	Copies BLOB or CLOB data into a file

For detailed information and the syntax of smart-large-object functions, see the Expression segment in the *Informix® Guide to SQL: Syntax*.

🚫 **Restriction:** Casts between BLOB and CLOB data types are not permitted.

Complex data types

A *complex data type* is usually a composite of other existing data types. For example, you might create a complex data type whose components include built-in types, opaque types, distinct types, or other complex types. An important advantage that complex data types have over user-defined types is that users can access and manipulate the individual components of a complex data type.

In contrast, built-in types and user-defined types are self-contained (encapsulated) data types. Consequently, the only way to access the component values of an opaque data type is through functions that you define on the opaque type.

The following figure shows the complex data types that supports and the syntax that you use to create the complex data types.

Figure 17. Complex Data Types
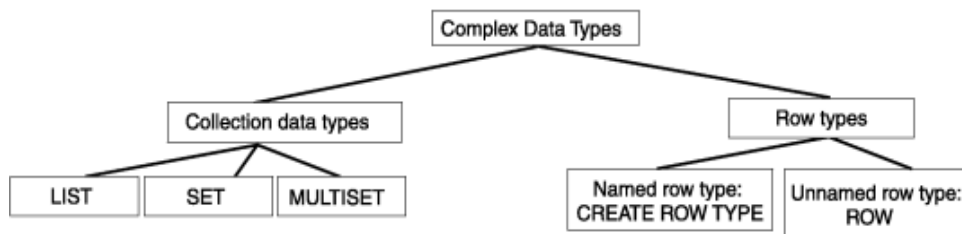
The complex data types that the previous figure illustrates provide the following extended data type support:

Collection types

> You can use a collection type whenever you must store and manipulate collections of data within a table cell. You can assign collection types to columns.

Row types

> A row type typically contains multiple fields. When you want to store more than one kind of data in a column or variable, you can create a row type. Row types come in two kinds: named row types and unnamed row types. You can assign an unnamed row type to columns and variables. You can assign a named row type to columns, variables, tables, or views. When you assign a named row type to a table, the table is a *typed table*. A primary advantage of typed tables is that they can be used to define an inheritance hierarchy.

For more information about how to perform SELECT, INSERT, UPDATE, and DELETE operations on the complex data types that this chapter describes, see the *Informix® Guide to SQL: Tutorial*.

Collection data types

Collection data types enable you to store and manipulate collections of data within a single row of a table. A collection data type has two components: a *type constructor*, which determines whether the collection type is a SET, MULTISET, or LIST, and

an *element type*, which specifies the type of data that the collection can contain. (The SET, MULTISET, and LIST collection types are described in detail in the following sections.)

The elements of a collection can be of most any data type. (For a list of exceptions, see Restrictions on collections on page 108.) The *elements* of a collection are the values that the collection contains. In a collection that contains the values: `{'blue', 'green', 'yellow', and 'red'}`, `'blue'` represents a single element in the collection. Every element in a collection must be of the same type. For example, a collection whose element type is INTEGER can contain only integer values.

The element type of a collection can represent a single data type (column) or multiple data types (row). In the following example, the **col_1** column represents a SET of integers:

```
col_1 SET(INTEGER NOT NULL)
```

To define a collection data type that contains multiple data types, you can use a named row type or an unnamed row type. In the following example, the **col_2** column represents a SET of rows that contain **name** and **salary** fields:

```
col_2 SET(ROW(name VARCHAR(20), salary INTEGER) NOT NULL)
```

> **Important:** When you define a collection data type, you must include the NOT NULL constraint as part of the type definition. No other column constraints are allowed on a collection data type.

After you define a column as a collection data type, you can perform the following operations on the collection:

- Select and modify individual elements of a collection (from Informix® ESQL/C programs only).
- Count the number of elements that a collection contains.
- Determine if certain values are in a collection.

For information about the syntax that you use to create collection data types, see the Data Type segment in the *Informix® Guide to SQL: Syntax*. For information about how to convert a value of one collection type to another collection type, see the *Informix® Guide to SQL: Tutorial*.

Null values in collections

A collection cannot contain NULL elements. However, when the collection is a row type, you can insert NULL values for any or all fields of a row type that a collection contains. Suppose you create the following table that has a collection column:

```
CREATE TABLE tab1 (col1 INT,
        col2 SET(ROW(a INT, b INT) NOT NULL));
```

The following statements are allowed because only the component fields of the row type specify NULL values:

```
INSERT INTO tab1 VALUES ( 25,"SET{ROW(NULL, NULL)}");

INSERT INTO tab1 VALUES ( 35,"SET{ROW(4, NULL)}");

INSERT INTO tab1 VALUES ( 45,"SET{ROW(14, NULL), ROW(NULL,5)}");

UPDATE tab1 SET col2 = "SET{ROW(NULL, NULL)}" WHERE col1 = 45;
```

However, each of the following statements returns an error message because the collection element specifies a NULL value:

```
INSERT INTO tab1 VALUES ( 45, "SET{NULL)}");

UPDATE tab1 SET col2 = "SET{NULL}" WHERE col1 = 55;
```

SET collection types

A SET is an unordered collection of elements in which each element is unique. You define a column as a SET collection type when you want to store collections whose elements have the following characteristics:

- The elements contain no duplicate values.
- The elements have no specific order associated with them.

To illustrate how you might use a SET, imagine that your human resources department requires information about the dependents of each employee in the company. You can use a collection type to define a column in an **employee** table that stores the names of an employee's dependents. The following statement creates a table in which the **dependents** column is defined as a SET:

```
CREATE TABLE employee
(
   name        CHAR(30),
   address     CHAR (40),
   salary      INTEGER,
   dependents  SET(VARCHAR(30) NOT NULL)
);
```

A query against the **dependents** column for any given row returns the names of all the dependents of the employee. In this case, SET is the correct collection type because the collection of dependents for each employee should not contain any duplicate values. A column that is defined as a SET ensures that each element in a collection is unique.

To illustrate how to define a collection type whose elements are a row type, suppose that you want the **dependents** column to include the name and birthdate of an employee's dependents. In the following example, the **dependents** column is defined as a SET whose element type is a row type:

```
CREATE TABLE employee
(
   name        CHAR(30),
   address     CHAR (40),
   salary      INTEGER,
   dependents  SET(ROW(name VARCHAR(30), bdate DATE) NOT NULL)
);
```

Each element of a collection from the **dependents** column contains values for the **name** and **bdate**. Each row of the **employee** table contains information about the employee and a collection with the names and birthdates of the employee's dependents. For example, if an employee has no dependents, the collection for the **dependents** column is empty. If an employee has 10 dependents, the collection should contain 10 elements.

MULTISET collection types

A MULTISET is a collection of elements in which the elements can have duplicate values. For example, a MULTISET of integers might contain the collection {1,3,4,3,3}, which has duplicate elements. You can define a column as a MULTISET collection type when you want to store collections whose elements have the following characteristics:

- The elements might not be unique.
- The elements have no specific order associated with them.

To illustrate how you might use a MULTISET, suppose that your human resources department wants to keep track of the bonuses awarded to employees in the company. To track each employee's bonuses over time, you can use a MULTISET to define a column in a table that records all the bonuses that each employee receives. In the following example, the **bonus** column is a MULTISET:

```
CREATE TABLE employee
(
    name      CHAR(30),
    address   CHAR (40),
    salary    INTEGER,
    bonus     MULTISET(MONEY NOT NULL)
);
```

You can use the **bonus** column in this statement to store and access the collection of bonuses for each employee. A query against the **bonus** column for any given row returns the dollar amount for each bonus that the employee has received. Because an employee might receive multiple bonuses of the same amount (resulting in a collection whose elements are not all unique), the **bonus** column is defined as a MULTISET, which allows duplicate values.

LIST collection types

A LIST is an ordered collection of elements that allows duplicate values. A LIST differs from a MULTISET in that each element in a LIST has an ordinal position in the collection. The order of the elements in a list corresponds with the order in which values are inserted into the LIST. You can define a column as a LIST collection type when you want to store collections whose elements have the following characteristics:

- The elements have a specific order associated with them.
- The elements might not be unique.

To illustrate how you might use a LIST, suppose your sales department wants to keep a monthly record of the sales total for each salesperson. You can use a LIST to define a column in a table that contains the monthly sales totals for each salesperson. The following example creates a table in which the **month_sales** column is a LIST. The first entry (element) in the LIST, with an ordinal position of 1, might correspond to the month of January, the second element, with an ordinal position of 2, February, and so forth:

```
CREATE TABLE sales_person
(
    name         CHAR(30),
    month_sales  LIST(MONEY NOT NULL)
);
```

You can use the **month_sales** column in this statement to store and access the monthly sales totals for each salesperson. More specifically, you might perform queries on the **month_sales** column to find out:

- The total sales that a salesperson generated during a specified month
- The total sales for every salesperson during a specified month

Nested collection types

A *nested collection* is a collection type that contains another collection type. You can nest any collection type within another collection type. There is no practical limit on how deeply you can nest a collection type. However, performing inserts or updates on a collection that has been nested more than one or two levels can be difficult.

The following example shows several ways in which you might create columns that are defined on nested collection types:

```
col_1 SET(MULTISET(VARCHAR(20) NOT NULL) NOT NULL);

col_2 MULTISET(ROW(x CHAR(5), y SET(INTEGER NOT NULL))
NOT NULL);

col_3 LIST(MULTISET(ROW(a CHAR(2), b INTEGER) NOT NULL)
NOT NULL);
```

For information about how to access a nested collection, see the *Informix® Guide to SQL: Tutorial*.

Add a collection type to an existing table

You can use the ALTER TABLE statement to add or drop a column that is a collection type (or any other data type). For example, the following statement adds the **flowers** column, which is defined as a SET, to the **nursery** table:

```
ALTER TABLE nursery ADD flower SET(VARCHAR(30) NOT NULL)
```

You cannot modify an existing column that is a collection type or convert a non-collection type column into a collection type.

For more information about adding and dropping collection-type columns, see the ALTER TABLE statement in the *Informix® Guide to SQL: Syntax*.

Restrictions on collections

You cannot use any of the following data types as the element type of a collection:

- TEXT
- BYTE
- SERIAL
- SERIAL8
- BIGSERIAL

You cannot use a CREATE INDEX statement to create an index on collection, nor can you create a functional index for a collection column.

Named row types

A *named row type* is a group of fields that are defined under a single name. A *field* refers to a component of a row type and should not be confused with a column, which is associated with tables only. The fields of a named row type are analogous to the fields of a C-language structure or members of a class in object-oriented programming. After you create a named row type, the name that you assign to the row type represents a unique type within the database. To create a named row type, you specify a name for the row type and the names and data types of its constituent fields. The following example shows how you might create a named row type called **person_t**:

```
CREATE ROW TYPE person_t
(
    name     VARCHAR(30) NOT NULL,
    address  VARCHAR(20),
    city     VARCHAR(20),
    state    CHAR(2),
    zip      VARCHAR(9),
    bdate    DATE
);
```

The **person_t** row type contains six fields: **name**, **address**, **city**, **state**, **zip**, and **bdate**. When you create a named row type, you can use it just as you would any other data type. The **person_t** can occur anywhere that you might use any other data type. The following CREATE TABLE statement uses the **person_t** data type:

```
CREATE TABLE sport_club
(
    sport    CHAR(20),
    sportnum INT,
    member   person_t,
    since    DATE,
    paidup   BOOLEAN
)
```

You can use most data types to define the fields of a row type. For information about data types that are not supported in row types, see Restrictions on named row types on page 110.

For the syntax you use to create a named row type, see the CREATE ROW TYPE statement in the *Informix® Guide to SQL: Syntax*. For information about how to cast row type values, see Create and use user-defined casts on page 129.

When to use a named row type

A named row type is one way to create a new data type in . When you create a named row type, you are defining a template for fields of data types known to the database server. Thus the field definitions of a row type are analogous to the column definitions of a table: both are constructed from data types known to the database server.

You can create a named row type when you want a type that acts as a container for component values that users must access. For example, you might create a named row type to support address values because users require direct access to

the individual component values of an address such as street, city, state, and zip code. When you create the address type as a named row type, users always have direct access to each of the fields.

In contrast, if you create an opaque data type to handle address values, a C-language data structure stores all the address information. Because the component values of an opaque type are encapsulated, you would have to define functions to extract the component values for street, city, state, zip code. Thus, an opaque data type is a more complicated type to define and use.

Before you define a data type, determine whether the type is just a container for a group of values that users can access directly. If the type fits this description, use a named row type.

Select a name for a named row type

You can give a named row type any name that you like provided that the name does not violate the conventions established for the SQL identifiers. The conventions for SQL identifiers are described in the Identifier segment in the *Informix® Guide to SQL: Syntax*. To avoid confusing type and table names, the examples in this manual designate named row types with the **_t** characters at the end of the row type name.

You must have the Resource privilege to create a named row type. The name that you assign to a named row type should not be the same as any other data type that exists in the database because all data types share the same name space. In an ANSI-compliant database, the combination `owner.type` must be unique within the database. In a database that is not ANSI-compliant, the name must be unique within the database.

⚠️ **Important:** You must grant USAGE privileges on a named row type before other users can use it.

Related information

[Usage privileges for user-defined types on page 68](#)

Restrictions on named row types

This section describes the restrictions that apply when you use named row types.

Restrictions on data types

You should use the BLOB or CLOB data types instead of the TEXT or BYTE data types when you create a typed table that contains columns for large objects. For compatibility with earlier version, you can create a named row type that contains TEXT or BYTE fields and use that type to re-create an existing (untyped) table as a typed table. However, although you can use a named row type that contains TEXT or BYTE fields to create a typed table, you cannot use such a row type as a column. You can assign a named row type that contains BLOB or CLOB fields to a typed table or column.

Restrictions on constraints

In a CREATE ROW TYPE statement, you can specify only the NOT NULL constraint for the fields of a named row type. You must define all other constraints in the CREATE TABLE statement. For more information, see the CREATE TABLE statement in the *Informix® Guide to SQL: Syntax*.

Restrictions on indexes

You cannot use a CREATE INDEX statement to create an index on a named row type column. However, you can use a user-defined routine to create a functional index for a row type column.

Restrictions on serial data types

A named row type that contains a SERIAL, SERIAL8, or BIGSERIAL data type cannot be used as a column type in a table.

The following statements return an error when the database server attempts to create the table:

```
CREATE ROW TYPE row_t (s_col SERIAL)

CREATE TABLE bad_tab (col1 row_t)
```

However, you can use a named row type that contains a SERIAL, SERIAL8, or BIGSERIAL data type to create a typed table.

For information about the use and behavior of SERIAL, SERIAL8, and BIGSERIAL types in table hierarchies, see SERIAL types in a table hierarchy on page 126.

Use a named row type to create a typed table

You can create a table that is typed or untyped. A *typed table* is a table that has a named row type assigned to it. An *untyped table* is a table that does not have a named row type assigned to it. The CREATE ROW TYPE statement creates a named row type but does not allocate storage for instances of the row type. To allocate storage for instances of a named row type, you must assign the row type to a table. The following example shows how to create a typed table:

```
CREATE ROW TYPE person_t
(
    name     VARCHAR(30),
    address  VARCHAR(20),
    city     VARCHAR(20),
    state    CHAR(2),
    zip      INTEGER,
    bdate    DATE
);

CREATE TABLE person OF TYPE person_t;
```

The first statement creates the **person_t** type. The second statement creates the **person** table, which contains instances of the **person_t** type. More specifically, each row in a typed table contains an instance of the named row type that is assigned to the table. In the preceding example, the fields of the **person_t** type define the columns of the **person** table.

> ⚠ **Important:** The order in which you create named row types is important because a named row type must exist before you can use it to define a typed table.

Inserting data into a typed table is no different than inserting data into an untyped table. When you insert data into a typed table, the operation creates an instance of the row type and inserts it into the table. The following example shows how to insert a row into the **person** table:

```
INSERT INTO person
VALUES ('Brown, James', '13 First St.', 'San Carlos', 'CA', 94070,
'01/04/1940')
```

The INSERT statement creates an instance of the **person_t** type and inserts it into the table. For more information about how to insert, update, and delete columns that are defined on named row types, see the *Informix® Guide to SQL: Tutorial*.

You can use a single named row type to create multiple typed tables. In this case, each table has a unique name, but all tables share the same type.

> 🚫 **Restriction:** You cannot create a typed table that is a temporary table.

For information about the advantages of using typed tables when you implement your data model, see Type inheritance on page 116.

Change the type of a table

The primary advantage of typed tables over untyped tables is that typed tables can be used in an inheritance hierarchy. In general, inheritance allows a table to acquire the representation and behavior of another table. For more information, see What is inheritance? on page 116.

The DROP and ADD clauses of the ALTER TABLE statement let you change between typed and untyped tables. Neither the ADD nor DROP operation affects the data that is stored in the table.

Convert an untyped table into a typed table

If you want to convert an existing untyped table into a typed table, you can use the ALTER TABLE statement. For example, consider the following untyped table:

```
CREATE TABLE manager
(
   name        VARCHAR(30),
   department  VARCHAR(20),
```

```
    salary      INTEGER
);
```

To convert an untyped table to a typed table, both the field names and the field types of the named row type must match the column names and column types of the existing table. For example, to make the **manager** table a typed table, you must first create a named row type that matches the column definitions of the table. The following statement creates the **manager_t** type, which contains field names and field types that match the columns of the **manager** table:

```
CREATE ROW TYPE manager_t
(
    name        VARCHAR(30),
    department  VARCHAR(20),
    salary      INTEGER
);
```

After you create the named row type that you want to assign to the existing untyped table, use the ALTER TABLE statement to assign the type to the table. The following statement alters the **manager** table and makes it a typed table of type **manager_t**:

```
ALTER TABLE manager ADD TYPE manager_t
```

The new **manager** table contains the same columns and data types as the old table but now provides the advantages of a typed table.

Convert a typed table into an untyped table

You also use the ALTER TABLE statement to change a typed table into an untyped table:

```
ALTER TABLE manager DROP TYPE
```

> **Tip:** Adding a column to a typed table requires three ALTER TABLE statements to drop the type, add the column, and add the type to the table.

Use a named row type to create a column

Both typed and untyped tables can contain columns that are defined on named row types. A column that is defined on a named row type behaves in the same way whether the column occurs in a typed table or untyped table. In the following example, the first statement creates a named row type **address_t**; the second statement assigns the **address_t** type to the **address** column in the **employee** table:

```
CREATE ROW TYPE address_t
(
    street  VARCHAR(20),
    city    VARCHAR(20),
    state   CHAR(2),
    zip     VARCHAR(9)
);

CREATE TABLE employee
(
```

```
    name     VARCHAR(30),
    address  address_t,
    salary   INTEGER
);
```

In the preceding CREATE TABLE statement, the **address** column has the **street**, **city**, **state**, and **zip** fields of the **address_t** type. Consequently, the **employee** table, which has only three columns, contains values for **name**, **street**, **city**, **state**, **zip**, and **salary**. Use dot notation to access the individual fields of a column that are defined on a row type. For information about using dot notation to access fields of a column, see the *Informix® Guide to SQL: Tutorial*.

When you insert data into a column that is assigned a row type, you must use the ROW constructor to specify row literal values for the row type. The following example shows how to use the INSERT statement to insert a row into the **employee** table:

```
INSERT INTO employee
VALUES ('John Bryant',
  ROW('10 Bay Street', 'Madera', 'CA', 95400)::address_t, 55000);
```

Strong typing is not enforced for an insert or update on a named row type. To ensure that the row values are of the named row type, you must explicitly cast to the named row type to generate values of a named row type, as the previous example shows. The INSERT statement inserts three values, one of which is a row type value that contains four values. More specifically, the operation inserts unitary values for the **name** and **salary** columns but it creates an instance of the **address_t** type and inserts it into the **address** column.

For more information about how to insert, update, and delete columns that are defined on row types, see the *Informix® Guide to SQL: Tutorial*.

Use a named row type within another row type

You can use a named row type as the data type of a field within another row type. A *nested row type* is a row type that contains another row type. You can nest any row type within any other row type. No practical limit exists on how deeply you can nest row types. However, to perform inserts or updates on deeply nested row types requires careful use of the syntax.

For named row types, the order in which you create the row types is important because a named row type must exist before you can use it to define a column or a field within another row type. In the following example, the first statement creates the **address_t** type, which is used in the second statement to define the type of the **address** field of the **employee_t** type:

```
CREATE ROW TYPE address_t
(
    street  VARCHAR (20),
    city    VARCHAR(20),
    state   CHAR(2),
    zip     VARCHAR(9)
);


CREATE ROW TYPE employee_t
(
    name     VARCHAR(30) NOT NULL,
    address  address_t,
    salary   INTEGER
);
```

⚠️ **Important:** You cannot use a row type recursively. If **type_t** is a row type, then you cannot use **type_t** as the data type of a field contained in **type_t**.

Drop named row types

To drop a named row type, use the DROP ROW TYPE statement. You can drop a type only if it has no dependencies. You cannot drop a named row type if any of the following conditions are true:

- The type is currently assigned to a table.
- The type is currently assigned to a column in a table.
- The type is currently assigned to a field within another row type.

The following example shows how to drop the **person_t** type:

```
DROP ROW TYPE person_t restrict;
```

For information about how to drop a named row type from a type hierarchy, see Drop named row types from a type hierarchy on page 120.

Unnamed row types

An *unnamed row type* is a group of typed fields that you create with the ROW constructor. An important distinction between named and unnamed row types is that you cannot assign an unnamed row type to a table. You use an unnamed row type to define the type of a column or field only. In addition, an unnamed row type is identified by its structure alone, whereas a named row type is identified by its name. The structure of a row type consists of the number and data types of its fields.

The following statement assigns two unnamed row types to columns of the **student** table:

```
CREATE TABLE student
(
   s_name  ROW(f_name VARCHAR(20), m_init CHAR(1),
            l_name VARCHAR(20) NOT NULL),
   s_address  ROW(street VARCHAR(20), city VARCHAR(20),
              state CHAR(2), zip VARCHAR(9))
   );
```

The **s_name** and **s_address** columns of the **student** table each contain multiple fields. Each field of an unnamed row type can have a different data type. Although the **student** table has only two columns, the unnamed row types define a total of seven fields:

- **f_name**
- **m_init**
- **l_name**
- **street**
- **city**

- **state**
- **zip**

The following example shows how to use the INSERT statement to insert data into the **student** table:

```
INSERT INTO student
VALUES (ROW('Jim', 'K', 'Johnson'), ROW('10 Grove St.',
'Eldorado', 'CA', 94108))
```

For more information about how to modify columns that are defined on row types, see the *Informix® Guide to SQL: Tutorial*.

The database server does not distinguish between two unnamed row types that contain the same number of fields and that have corresponding fields of the same type. Field names are irrelevant in type checking of unnamed row types. For example, the database server does not distinguish between the following unnamed row types:

```
ROW(a INTEGER, b CHAR(4));
ROW(x INTEGER, y CHAR(4));
```

For the syntax of unnamed row types, see the *Informix® Guide to SQL: Syntax*. For information about how to cast row type values, see Create and use user-defined casts on page 129.

The following data types cannot be field types in an unnamed row type:

- BIGSERIAL
- SERIAL
- SERIAL8
- BYTE
- TEXT

The database server returns an error when any of the preceding types are specified in the field definition of an unnamed row type.

Type and table inheritance

This chapter describes type and table inheritance and shows how to create type and table hierarchies to modify the types and tables within the respective hierarchies.

What is inheritance?

Inheritance is the process that allows a type or a table to acquire the properties of another type or table. The type or table that inherits the properties is called the *subtype* or *subtable*. The type or table whose properties are inherited is called the *supertype* or *supertable*. Inheritance allows for incremental modification so that a type or table can inherit a general set of properties and add properties that are specific to itself. You can use inheritance to make modifications only to the extent that the modifications do not alter the inherited supertypes or supertables.

supports inheritance only for named row types and typed tables. Informix® supports only single inheritance. With *single inheritance*, each subtype or subtable has only one supertype or supertable.

Type inheritance

Type inheritance applies to named row types only. You can use inheritance to group named row types into a *type hierarchy* in which each subtype inherits the representation (data fields) and the behavior (UDRs, aggregates, and operators) of the supertype under which it is defined. A type hierarchy provides the following advantages:

- It encourages modular implementation of your data model.
- It ensures consistent reuse of schema components.
- It ensures that no data fields are accidentally left out.
- It allows a type to inherit UDRs that are defined on another data type.

Define a type hierarchy

The following figure provides an example of a simple type hierarchy that contains three named row types.

Figure 18. Example of a type hierarchy
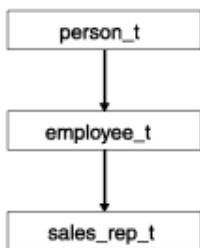
The supertype at the top of the type hierarchy contains a group of fields that all underlying subtypes inherit. A supertype must exist before you can create its subtype. The following example creates the **person_t** supertype of the type hierarchy that shows:

```
CREATE ROW TYPE person_t
(
    name     VARCHAR(30) NOT NULL,
    address  VARCHAR(20),
    city     VARCHAR(20),
    state    CHAR(2),
    zip      INTEGER,
    bdate    DATE
);
```

To create a subtype, specify the UNDER keyword and the name of the supertype whose properties the subtype inherits. The following example illustrates how you might define **employee_t** as a subtype that inherits all the fields of **person_t**. The example adds **salary** and **manager** fields that do not exist in the **person_t** type.

```
CREATE ROW TYPE employee_t
(
    salary   INTEGER,
    manager  VARCHAR(30)
)
UNDER person_t;
```

> ⚠️ **Important:** You must have the UNDER privilege on the supertype before you can create a subtype that inherits the properties of the supertype.

In the type hierarchy in , **sales_rep_t** is a subtype of **employee_t**, which is the supertype of **sales_rep_t** in the same way that **person_t** is the supertype of **employee_t**. The following example creates **sales_rep_t**, which inherits all fields from **person_t** and **employee_t** and adds four new fields. Because the modifications on a subtype do not affect its supertype, **employee_t** does not have the four fields that are added for **sales_rep_t**.

```
CREATE ROW TYPE sales_rep_t
(
rep_num      INT8,
region_num   INTEGER,
commission   DECIMAL,
home_office  BOOLEAN
)
UNDER employee_t;
```

The **sales_rep_t** type contains 12 fields: **name**, **address**, **city**, **state**, **zip**, **bdate**, **salary**, **manager**, **rep_num**, **region_num**, **commission**, and **home_office**.

Instances of both the **employee_t** and **sales_rep_t** types inherit all the UDRs that are defined for the **person_t** type. Any additional UDRs that are defined on **employee_t** automatically apply to instances of the **employee_t** type and to instances of its subtype **sales_rep_t**, but not to instances of **person_t**.

The preceding type hierarchy is an example of single inheritance because each subtype inherits from a single supertype. illustrates how you can define multiple subtypes under a single supertype. Although single inheritance requires that every subtype inherits from one and only one supertype, no practical limit exists on the depth or breadth of the type hierarchy that you define.

Figure 19. Example of a type hierarchy that is a tree structure
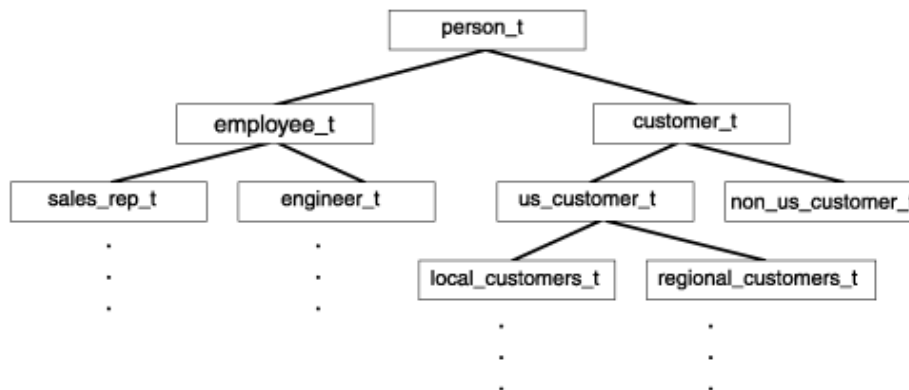
The topmost type of any hierarchy is referred to as the *root supertype*. In , **person_t** is the root supertype of the hierarchy. Except for the root supertype, any type in the hierarchy can be potentially both a supertype and subtype at the same time. For example, **customer_t** is a subtype of **person_t** and a supertype of **us_customer_t**. A subtype at the lower levels of the hierarchy contains properties of the root supertype but does not directly inherit its properties from the root supertype. For example, **us_customer_t** has only one

supertype, **customer_t**, but because **customer_t** is itself a subtype of **person_t**, the fields and routines that **customer_t** inherits from **person_t** are also inherited by **us_customer_t**.

Related information

Routine overloading for types in a type hierarchy

Routine overloading refers to the ability to assign one name to multiple routines and specify different types of arguments on which the routines can operate. In a type hierarchy, a subtype automatically inherits the routines that are defined on its supertype. However you can define a new routine on a subtype to override the inherited routine with the same name. For example, suppose you create a **getinfo()** routine on type **person_t** that returns the last name and birthdate of an instance of type **person_t**. You can register another **getinfo()** routine on type **employee_t** that returns the last name and salary from an instance of **employee_t**. In this way, you can overload a routine, so that you have a customized routine for every type in the type hierarchy, as the following figure shows.

Figure 20. Example of routine overloading in a type hierarchy
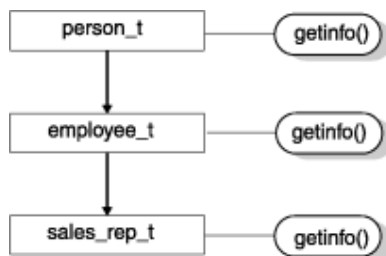
When you overload a routine so that routines are defined with the same name but different arguments for different types in the type hierarchy, the argument that you specify determines which routine executes. For example, if you call **getinfo()** with an argument of type **employee_t**, a **getinfo()** routine defined on type **employee_t** overrides the inherited routine of the same name. Similarly, if you define another **getinfo()** on type **sales_rep_t**, a call to **getinfo()** with an argument of type **sales_rep_t** overrides the routine that **sales_rep_t** inherits from **employee_t**.

For information about how to create and register user-defined routines (UDRs), see *Informix® User-Defined Routines and Data Types Developer's Guide*.

Inheritance and type substitutability

In a type hierarchy, a subtype automatically inherits all the routines defined on its supertype. Consequently, if you call a routine with an argument of a subtype and no routines are defined on the subtype, the database server can invoke a routine that is defined on a supertype. *Type substitutability* refers to the ability to use an instance of a subtype when an instance of a supertype is expected. As an example, suppose that you create a routine **p_info()** that accepts an argument of type **person_t** and returns the last name and birthdate of an instance of type **person_t**. If no other **p_info()** routines are registered, and

you invoke **p_info()** with an argument of type **employee_t**, the routine returns the name and birthdate fields (inherited from **person_t**) from an instance of type **employee_t**. This behavior is possible because **employee_t** inherits the functions of its supertype, **person_t**.

In general, when the database server attempts to evaluate a routine, the database server searches for a signature that matches the routine name and the arguments that you specify when you invoke the routine. If such a routine is found, then the database server uses this routine. If an exact match is not found, the database server attempts to find a routine with the same name and whose argument type is a supertype of the argument type that is specified when the routine is invoked. For example, suppose that the database server searches for a routine that it can use when a **get()** routine is called with an argument of the subtype **sales_rep_t**. Although no **get()** routine has been defined on the **sales_rep_t** type, the database server searches for a routine until it finds a **get()** routine that has been defined on a supertype in the hierarchy. In this case, neither **sales_rep_t** nor its supertype **employee_t** has a **get()** routine defined over it. However, because a routine is defined for **person_t**, this routine is invoked to operate on an instance of **sales_rep_t**.

Figure 21. Example of how the database server searches for a routine in a type hierarchy
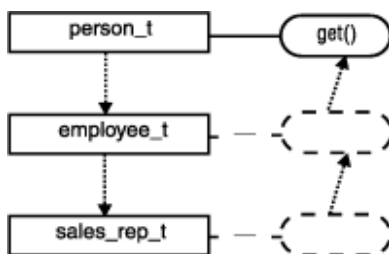
The process in which the database server searches for a routine that it can use is called *routine resolution*. For more information about routine resolution, see *Informix® User-Defined Routines and Data Types Developer's Guide*.

Drop named row types from a type hierarchy

To drop a named row type from a type hierarchy, use the DROP ROW TYPE statement. However, you can drop a type only if it has no dependencies. You cannot drop a named row type if either of the following conditions is true:

- The type is currently assigned to a table.
- The type is a supertype of another type.

The following example shows how to drop the **sales_rep_t** type:

```
DROP ROW TYPE sales_rep_t RESTRICT;
```

To drop a supertype, you must first drop each subtype that inherits properties from the supertype. You drop types in a type hierarchy in the reverse order in which you create the types. For example, to drop the **person_t** type that Inheritance and type substitutability on page 119 shows, you must first drop its subtypes in the following order:

```
DROP ROW TYPE sale_rep_t RESTRICT;
DROP ROW TYPE employee_t RESTRICT;
DROP ROW TYPE person_t RESTRICT;
```

> ⚠️ **Important:** To drop a type, you must be the database administrator or the owner of the type.

Table inheritance

Only tables that are defined on named row types support *table inheritance*. Table inheritance is the property that allows a table to inherit the behavior (constraints, storage options, triggers) from the supertable above it in the *table hierarchy*. A table hierarchy is the relationship that you can define among tables in which subtables inherit the behavior of supertables. A table inheritance provides the following advantages:

- It encourages modular implementation of your data model.
- It ensures consistent reuse of schema components.
- It allows you to construct queries whose scope can be some or all of the tables in the table hierarchy.

In a table hierarchy, a subtable automatically inherits the following properties from its supertable:

- All constraint definitions (primary key, unique, and referential constraints)
- Storage option
- All triggers
- Indexes
- Access method

Relationship between type and table hierarchies

Every table in a table hierarchy must be assigned to a named row type in a corresponding type hierarchy. The following figure shows an example of the relationships that can exist between a type hierarchy and table hierarchy.

Figure 22. Example of the relationship between type hierarchy and table hierarchy
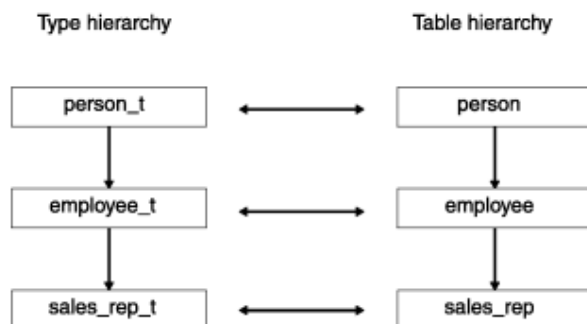
However, you can also define a type hierarchy in which the named row types do not necessarily have a one-to-one correspondence with the tables in a table hierarchy. The following figure shows how you might create a type hierarchy for which only some of the named row types have been assigned to tables.

Figure 23. Example of an inheritance hierarchy in which only some types have been assigned to tables
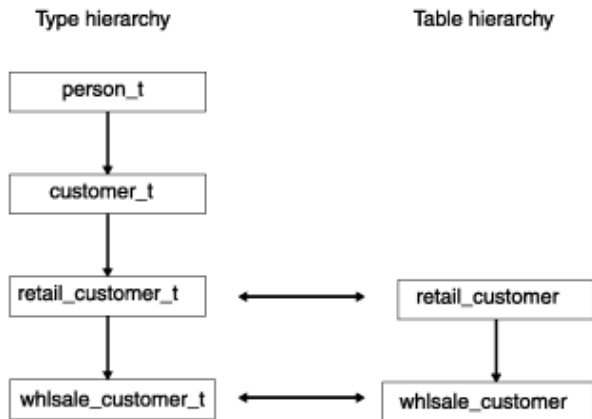
Define a table hierarchy

The type that you use to define a table must exist before you can create the table. Similarly, you define a type hierarchy before you define a corresponding table hierarchy. To establish the relationships between specific subtables and supertables in a table hierarchy, use the UNDER keyword. The following CREATE TABLE statements define the simple table hierarchy that Figure 22: Example of the relationship between type hierarchy and table hierarchy on page 121 shows. The examples in this section assume that the **person_t**, **employee_t**, and **sales_rep_t** types already exist.

```
CREATE TABLE person OF TYPE person_t;

CREATE TABLE employee OF TYPE employee_t UNDER person;

CREATE TABLE sales_rep OF TYPE sales_rep_t UNDER employee;
```

The **person**, **employee**, and **sales_rep** tables are defined on the **person_t**, **employee_t**, and **sales_rep_t** types, respectively. Thus, for every type in the type hierarchy, a corresponding table exists in the table hierarchy. In addition, the relationship between the tables of a table hierarchy must match the relationship between the types of the type hierarchy. For example, the **employee** table inherits from **person** table in the same way that the **employee_t** type inherits from the **person_t** type, and the **sales_rep** table inherits from the **employee** table in the same way that the **sales_rep_t** type inherits from the **employee_t** type.

Subtables automatically inherit all inheritable properties that are added to supertables. Therefore, you can add or alter the properties of a supertable at any time and the subtables automatically inherit the changes. For more information, see Modify table behavior in a table hierarchy on page 124.

⚠️ **Important:** You must have the UNDER privilege on the supertable before you can create a subtable that inherits the properties of the supertable. For more information, see .

Inheritance of table behavior in a table hierarchy

When you create a subtable under a supertable, the subtable inherits all the properties of its supertable, including the following ones:

- All columns of the supertable
- Constraint definitions
- Storage options
- Indexes
- Referential integrity
- Triggers
- The access method

In addition, if table **c** inherits from table **b** and table **b** inherits from table **a**, then table **c** automatically inherits the behavior unique to table **b** and the behavior that table **b** has inherited from table **a**. Consequently, the supertable that actually defines behavior can be several levels distant from the subtables that inherit the behavior. For example, consider the following table hierarchy:

```
CREATE TABLE person OF TYPE person_t
(PRIMARY KEY (name))
FRAGMENT BY EXPRESSION
name < 'n' IN dbspace1,
name >= 'n' IN dbspace2;

CREATE TABLE employee OF TYPE employee_t
(CHECK(salary > 34000))
UNDER person;

CREATE TABLE sales_rep OF TYPE sales_rep_t
LOCK MODE ROW
UNDER employee;
```

In this table hierarchy, the **employee** and **sales_rep** tables inherit the primary key name and fragmentation strategy of the **person** table. The **sales_rep** table inherits the check constraint of the **employee** table and adds a LOCK MODE. The following table shows the behavior for each table in the hierarchy.

Table

    Table Behavior

person

    PRIMARY KEY, FRAGMENT BY EXPRESSION

employee

    PRIMARY KEY, FRAGMENT BY EXPRESSION, CHECK constraint

sales_rep

    PRIMARY KEY, FRAGMENT BY EXPRESSION, CHECK constraint, LOCK MODE ROW

A table hierarchy might also contain subtables in which behavior defined on a subtable can override behavior (otherwise) inherited from its supertable. Consider the following table hierarchy, which is identical to the previous example except that the **employee** table adds a new storage option:

```
CREATE TABLE person OF TYPE person_t
(PRIMARY KEY (name))
FRAGMENT BY EXPRESSION
name < 'n' IN person1,
name >= 'n' IN person2;

CREATE TABLE employee OF TYPE employee_t
(CHECK(salary > 34000))
FRAGMENT BY EXPRESSION
name < 'n' IN employ1,
name >= 'n' IN employ2
UNDER person;

CREATE TABLE sales_rep OF TYPE sales_rep_t
LOCK MODE ROW
UNDER employee;
```

Again, the **employee** and **sales_rep** tables inherit the primary key name of the **person** table. However, the fragmentation strategy of the **employee** table overrides the fragmentation strategy of the **person** table. Consequently, both the **employee** and **sales_rep** tables store data in dbspaces **employ1** and **employ2**, whereas the **person** table stores data in dbspaces **person1** and **person2**.

Modify table behavior in a table hierarchy

After you define a table hierarchy, you cannot modify the structure (columns) of the existing tables. However, you can modify the behavior of tables in the hierarchy. Table 4: Table behavior that you can modify in a table hierarchy on page 124 shows the table behavior that you can modify in a table hierarchy and the syntax that you use to make modifications.

Table 4. Table behavior that you can modify in a table hierarchy

Table behavior	Syntax	Considerations
Constraint definitions	ALTER TABLE	To add or drop a constraint, use the ADD CONSTRAINT or DROP CONSTRAINT clause. For more information, see Constraints on tables in a table hierarchy on page 125.
Indexes	CREATE INDEX, ALTER INDEX	For more information, see Add indexes to tables in a table hierarchy on page 125 and the CREATE INDEX and ALTER INDEX statements in the *Informix® Guide to SQL: Syntax*.
Triggers	CREATE/DROP TRIGGER	You cannot drop an inherited trigger. However, you can drop a trigger from a supertable or add a trigger to a

Table 4. Table behavior that you can modify in a table hierarchy (continued)

Table behavior	Syntax	Considerations
		subtable to override an inherited trigger. For information about how to modify triggers on supertables and subtables, see Triggers on tables in a table hierarchy on page 125. For information about how to create a trigger, see the *Informix® Guide to SQL: Tutorial*.

All existing subtables automatically inherit new table behavior when you modify a supertable in the hierarchy.

Important: When you use the ALTER TABLE statement to modify a table in a table hierarchy, you can use only the ADD CONSTRAINT, DROP CONSTRAINT, MODIFY NEXT SIZE, and LOCK MODE clauses.

Constraints on tables in a table hierarchy

You can alter or drop a constraint only in the table on which it is defined. You cannot drop or alter a constraint from a subtable when the constraint is inherited. However, a subtable can add additional constraints. Any additional constraints that you define on a table are also inherited by any subtables that inherit from the table that defines the constraint. Because constraints are additive, all inherited and current (added) constraints apply.

Add indexes to tables in a table hierarchy

When you define an index on a supertable in a hierarchy, any subtables that you define under that supertable also inherit the index. Suppose you have a table hierarchy that contains the tables **tab_a**, **tab_b**, and **tab_c** where **tab_a** is a supertable to **tab_b**, and **tab_b** is a supertable to **tab_c**. If you create an index on a column of **tab_b**, then that index will exist on that column in both **tab_b** and **tab_c**. If you create an index on a column of **tab_a**, then that index will span **tab_a**, **tab_b**, and **tab_c**.

Important: An index that a subtable inherits from a supertable cannot be dropped or modified. However, you can add indexes to a subtable.

Indexes, unique constraints, and primary keys are all closely related. When you specify a unique constraint or primary key, the database server automatically creates a unique index on the column. Consequently, a primary key or unique constraint that you define on a supertable applies to all the subtables. For example, suppose there are two tables (a supertable and subtable), both of which contain a column **emp_id**. If the supertable specifies that **emp_id** has a unique constraint, the subtable must contain **emp_id** values that are unique across both the subtable and the supertable.

Restriction: You cannot define more than one primary key across a table hierarchy, even if some of the tables in the hierarchy do not inherit the primary key.

Triggers on tables in a table hierarchy

You cannot drop an inherited trigger. However, you can create a trigger on a subtable to override a trigger that the subtable inherits from a supertable. Unlike constraints, triggers are not additive; only the nearest trigger on a supertable in the hierarchy applies.

If you want to disable the trigger that a subtable inherits from its supertable, you can create an empty trigger on the subtable to override the trigger from the supertable. Because triggers are not additive, this empty trigger executes for the subtable and any subtables under the subtable, which are not subject to further overrides.

SERIAL types in a table hierarchy

A table hierarchy can contain columns of type SERIAL and BIGSERIAL or SERIAL8. However, only one SERIAL and one BIGSERIAL or one SERIAL8 column are allowed across a table hierarchy. Suppose you create the following type and table hierarchy:

```
CREATE ROW TYPE parent_t (a INT);
CREATE ROW TYPE child1_t (s_col SERIAL) UNDER parent_t;
CREATE ROW TYPE child2_t (s8_col SERIAL8) UNDER child1_t;
CREATE ROW TYPE child3_t (d FLOAT) UNDER child2_t;

CREATE TABLE parent_tab of type parent_t;
CREATE TABLE child1_tab of type child1_t UNDER parent_tab;
CREATE TABLE child2_tab of type child2_t UNDER child1_tab;
CREATE TABLE child3_tab of type child3_t UNDER child2_tab;
```

The **parent_tab** table does not contain a SERIAL type. The **child1_tab** introduces a SERIAL counter into the hierarchy. The **child2_tab** inherits the SERIAL column from **child1_tab** and adds a SERIAL8 column. The **child3_tab** inherits both a SERIAL and SERIAL8 column.

A 0 value inserted into the **s_col** or **s8_col** column for any table in the hierarchy inserts a monotonically increasing value, regardless of which table takes the insert.

You cannot set a starting counter value for a SERIAL or SERIAL8 type in CREATE ROW TYPE statements. To set a starting value for a SERIAL or SERIAL8 column in a table hierarchy, you can use the ALTER TABLE statement. The following statement shows how to alter a table to modify the next SERIAL and SERIAL8 values to be inserted anywhere in the table hierarchy:

```
ALTER TABLE child3_tab
MODIFY (s_col SERIAL(100), s8_col SERIAL8 (200))
```

Except for the previously described behavior, all the rules that apply to SERIAL, BIGSERIAL, and SERIAL8 type columns in untyped tables also apply to SERIAL, BIGSERIAL, and SERIAL8 type columns in table hierarchies. For more information, see Select data types on page 21 and the *Informix® Guide to SQL: Reference*.

Add a new table to a table hierarchy

After you define a table hierarchy, you cannot use the ALTER TABLE statement to add, drop, or modify columns of a table within the hierarchy. However, you can add new subtypes and subtables to an existing hierarchy provided that the new subtype and subtable do not interfere with existing inheritance relationships. The following figure illustrates one way that

you might add a type and corresponding table to an existing hierarchy. The dashed lines indicate the added subtype and subtable.

Figure 24. Example of how you might add a subtype and subtable to an existing inheritance hierarchy
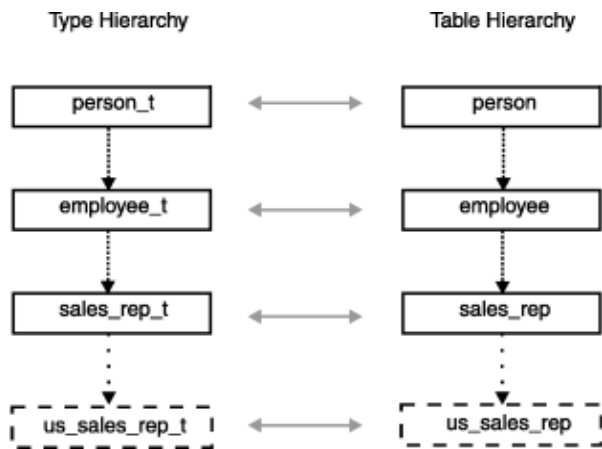
The following statements show how you might add the type and table to the inheritance hierarchy that Figure 24: Example of how you might add a subtype and subtable to an existing inheritance hierarchy on page 127 shows:

```
CREATE ROW TYPE us_sales_rep_t (domestic_sales DECIMAL(15,2))
UNDER employee_t;

CREATE TABLE us_sales_rep OF TYPE us_sales_rep_t
UNDER sales_rep;
```

You can also add subtypes and subtables that branch from an existing supertype and its parallel supertable. The following figure shows how you might add the **customer_t** type and **customer** table to existing hierarchies. In this example, both the **customer** table and the **employee** table inherit properties from the **person** table.

Figure 25. Example of Adding a Type and Table Under an Existing Supertype and Supertable
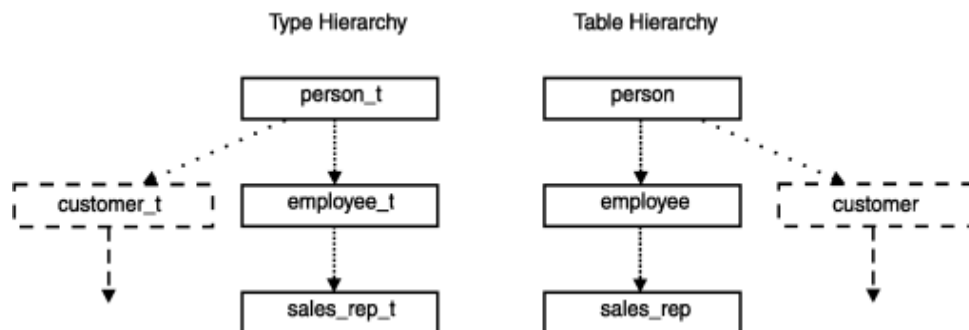
The following statements create the **customer_t** type and **customer** table under the **person_t** type and **person** table, respectively:

```
CREATE ROW TYPE customer_t (cust_num INTEGER) UNDER person_t;

CREATE TABLE customer OF TYPE customer_t UNDER person;
```

Drop a table in a table hierarchy

If a table and its corresponding named row type have no dependencies (they are not a supertable and supertype), you can drop the table and its type. You must drop the table before you can drop the type. For general information about dropping a table, see the DROP TABLE statement in the *Informix® Guide to SQL: Syntax*. For information about how to drop a named row type, see Drop named row types on page 115.

Altering the structure of a table in a table hierarchy

About this task

You cannot use the ALTER TABLE statement to add, drop, or modify the columns of a table in a table hierarchy. You can use the ALTER TABLE statement to add, drop, or modify constraints.

The process of adding, dropping, or modifying a column of a table in a table hierarchy (or otherwise altering the structure of a table) can be a time-intensive task.

To alter the structure of a table in a table hierarchy:

1. Download data from all subtables and the supertable that you want to modify.
2. Drop the subtables and subtypes.
3. Modify the unloaded data file.
4. Modify the supertable.
5. Recreate the subtypes and subtables.
6. Upload the data.

Query tables in a table hierarchy

A table hierarchy allows you to construct a SELECT, UPDATE, or DELETE statement whose scope is a supertable and its subtables-in a single SQL command. For example, a query against any supertable in a table hierarchy returns data for all columns of the supertable and the columns that subtables inherit from the supertable. To limit the results of a query to one table in the table hierarchy, you must include the ONLY keyword in the query. For more information about how to query and modify data from tables in a table hierarchy, see the *Informix® Guide to SQL: Tutorial*.

Create a view on a table in a table hierarchy

You can create a view based upon any table in a table hierarchy. For example, the following statement creates a view on the **person** table, which is the root supertable of the table hierarchy that Figure 22: Example of the relationship between type hierarchy and table hierarchy on page 121 shows:

```
CREATE VIEW name_view AS SELECT name FROM person
```

Because the **person** table is a supertable, the view **name_view** displays data from the **name** column of the **person**, **employee**, and **sales_rep** tables. To create a view that displays only data from the **person** table, use the ONLY keyword, as the following example shows:

```
CREATE VIEW name_view AS SELECT name FROM ONLY(person)
```

🚫 **Restriction:** You cannot perform an insert or update on a view that is defined on a supertable because the database server cannot know where in the table hierarchy to put the new rows.

For information about how to create a typed view, see .

Create and use user-defined casts

This chapter describes user-defined casts and shows how to use run-time casts to perform data conversions on extended data types.

What is a cast?

A *cast* is a mechanism that converts a value from one data type to another data type. Casts allow you to make comparisons between values of different data types or substitute a value of one data type for a value of another data type. supports casts in the following types of expressions:

- Column expressions
- Constant expressions
- Function expressions
- SPL variables
- Host variables (ESQL)
- Statement local variable (SLV) expressions

To convert a value of one data type to another data type, a cast must exist in the database or the database server. Informix® supports the following types of casts:

Built-in cast

A built-in cast is a cast that is built into the database server. A built-in cast performs automatic conversions between different built-in data types.

User-defined cast

A user-defined cast often requires a *cast function* to handle conversions from one data type to another. To register and use a user-defined cast, you must use the CREATE CAST statement.

A user-defined cast is *explicit* if you include the EXPLICIT keyword when you create a cast with the CREATE CAST statement. (The default option is explicit.) Explicit casts are never invoked automatically. To invoke an explicit cast, you must use the CAST... AS keywords or the double colon (::) cast operator.

A user-defined cast is *implicit* if you include the IMPLICIT keyword when you create a cast with a CREATE CAST statement. The database server automatically invokes implicit casts at runtime to perform data conversions.

All casts are included in the **syscasts** system catalog table. For information about **syscasts**, see the *Informix® Guide to SQL: Reference*.

User-defined casts

When the database server does not provide built-in casts to perform conversions between two data types, you can create a user-defined cast to handle the data type conversion. User-defined casts are typically used to provide data type conversions for the following extended data types:

Opaque data types

Developers of opaque data types must define casts to handle conversions between the internal/external representations of the opaque data type. For information about how to create and register casts for opaque data types, see *Informix® User-Defined Routines and Data Types Developer's Guide*.

Distinct data types

You cannot directly compare a distinct data type to its source type. However, automatically registers explicit casts from the distinct type to the source type and vice versa. A distinct type does not inherit the casts that are defined on its source type. In addition, the user-defined casts that you might define on a distinct type are unavailable to its source type. For more information and examples that show how to create and use casts on distinct types, see Create cast functions for user-defined casts on page 139.

Named row types

In most cases, you can explicitly cast a named row type to another row-type value without creating the cast. However, to convert between values of a named row type and some other data type, you must first create the cast to handle the conversion.

For an example of how to create and use a user-defined cast, see An example of casting between distinct data types on page 140. For the syntax of the CREATE CAST statement, see the *Informix® Guide to SQL: Syntax*.

Invoke casts

For built-in casts and user-defined implicit casts, the database server automatically (implicitly) invokes the cast to handle the data conversion. For example, you can compare a value of type INT with SMALLINT, FLOAT, or CHAR values without explicitly casting the expression because the database server provides system-defined casts to transparently handle conversions between these built-in data types.

When you define an explicit user-defined cast to handle conversions between two data types, you must explicitly invoke the cast with either the CAST...AS keywords or the double-colon cast operator (::). The following partial examples show the two ways that you can invoke an explicit cast:

```
... WHERE new_col = CAST(old_col AS newtype)

... WHERE new_col = old_col::newtype
```

Restrictions on user-defined casts

You cannot create a user-defined cast between two built-in data types. You also cannot create a user-defined cast that includes any of the following data types:

- Collection data types: LIST, MULTISET, or SET
- Unnamed row types
- Smart-large-object data types: CLOB or BLOB
- Simple-large-object data types: TEXT or BYTE

In general, a cast between two data types requires that each data type represents the same number of component values. For example, a cast between a row type and an opaque data type is possible if each field in the row type has a corresponding field in the opaque data type. When you want to perform conversions between two data types that have the same storage structure, you can use the CREATE CAST statement without a cast function. Otherwise, you must create a cast function that you then register with a CREATE CAST statement. For an example of how to use a cast function to create a user-defined cast, see Create cast functions for user-defined casts on page 139.

Cast row types

You can compare or substitute between values of any two row types (named or unnamed) only when both row types have the same number of fields, and one of the following conditions is also true:

- All corresponding fields of the two row types have the same data type.

  Two row types are considered *structurally equivalent* when they have the same number of fields and the data types of corresponding fields are the same.
- User-defined casts exist to perform the conversions when two named row types are being compared.
- System-defined or user-defined casts exist to perform the necessary conversions for corresponding field values that are not of the same data type.

  When the corresponding fields are not of the same data type, you can use either system-defined casts or user-defined casts to handle data conversions on the fields.

If a built-in cast exists to handle data conversions on the individual fields, you can explicitly cast the value of one row type to the other row type (unless the row types are both unnamed row types, in which case an explicit cast is not necessary).

If a built-in cast does not exist to handle field conversions, you can create a user-defined cast to handle the field conversions. The cast can be either implicit or explicit.

In general, when a row type is cast to another row type, the individual field conversions might be handled with explicit or implicit casts. When the conversion between corresponding fields requires an explicit cast, the value of the field that is cast

must match the value of the corresponding field exactly, because the database server applies no additional implicit casts on a value that has been explicitly cast.

Cast between named and unnamed row types

To compare values of a named row type with values of an unnamed row type, you can use an explicit cast. Suppose that you create the following named row type and tables:

```
CREATE ROW TYPE info_t (x CHAR(1), y CHAR(20))
CREATE TABLE customer (cust_info info_t)
CREATE TABLE retailer (ret_info ROW (a CHAR(1), b CHAR(20)))
```

The following INSERT statements show how to create row-type values for the **customer** and **retailer** tables:

```
INSERT INTO customer VALUES(ROW('t','philips')::info_t)
INSERT INTO retailer VALUES(ROW('f','johns'))
```

To compare or substitute data from the **customer** table with data from **retailer** table, you must use an explicit cast to convert a value of one row type to the other row type. In the following query, the **ret_info** column (an unnamed row type) is explicitly cast to **info_t** (a named row type):

```
SELECT cust_info
FROM customer, retailer
WHERE cust_info = ret_info::info_t
```

In general, to perform a conversion between a named row type and an unnamed row type, you must explicitly cast one row type to the other row type. You can perform an explicit cast in either direction: you can cast the named row type to an unnamed row type or cast the unnamed row type to a named row type. The following statement returns the same results as the previous example. However, the named row type in this example is explicitly cast to the unnamed row type:

```
SELECT cust_info
FROM customer, retailer
WHERE cust_info::ROW(a CHAR(1), b CHAR(20)) = ret_info
```

Cast between unnamed row types

You can compare two unnamed row types that are structurally equivalent without an explicit cast. You can also compare an unnamed row type with another unnamed row type, if both row types have the same number of fields, and casts exist to convert values of corresponding fields that are not of the same data type. In other words, the cast from one unnamed row type to another is implicit if all the casts that handle field conversions are system-defined or implicit casts. Otherwise, you must explicitly cast an unnamed row type to compare it with another row type.

Suppose you create the following **prices** table:

```
CREATE TABLE prices
(col1  ROW(a SMALLINT, b FLOAT)
 col2  ROW(x INT, y REAL) )
```

The values of the two unnamed row types can be compared (without an explicit cast) when built-in casts exist to perform conversions between corresponding fields. Consequently, the following query does not require an explicit cast to compare **col1** and **col2** values:

```
SELECT * FROM prices WHERE col1 = col2
```

In this example, the database server implicitly invokes a built-in cast to convert field values of SMALLINT to INT and REAL to FLOAT.

If corresponding fields of two row types cannot implicitly cast to one another, you can explicitly cast between the types, if a user-defined cast exists to handle conversions between the two types.

Cast between named row types

A named row type is strongly typed, which means that the database server recognizes two named row types as two separate types even if the row types are structurally equivalent. For this reason you must create and register a user-defined cast before you can perform comparisons between two named row types. For an example of how to create and use casts to handle conversions between two named row types, see An example of casting between named row types on page 139.

Explicit casts on fields

Before you can explicitly cast between two row types (named or unnamed), whose fields contain different data types, a cast (either system-defined or user-defined) must exist to handle conversions between the corresponding field data types.

When you explicitly cast between two row types, the database server automatically invokes any explicit casts that are necessary to handle conversions between field data types. In other words, when you perform an explicit cast on a row type value, you do not have to explicitly cast individual fields of the row type, unless more than one level of casting is necessary to handle the data type conversion on the field.

The row types and tables in the following example are used throughout this section to show the behavior of explicit casts on named and unnamed row types:

```
CREATE DISTINCT TYPE d_float AS FLOAT;
CREATE ROW TYPE row_t (a INT, b d_float);

CREATE TABLE tab1 (col1 ROW (a INT, b d_float));
CREATE TABLE tab2 (col2 ROW (a INT, b FLOAT));
CREATE TABLE tab3 (col3 row_t);
```

Explicit casts on fields of an unnamed row type

When a conversion between two row types involves an explicit cast to convert between particular field values, you can explicitly cast the row type value but are not required to explicitly cast the individual field.

The following statement shows how to insert a value into the **tab1** table:

```
INSERT INTO tab1 VALUES (ROW( 3, 5.66::FLOAT::d_float))
```

To insert a value from **col1** of **tab1** into **col2** of **tab2**, you must explicitly cast the row value because the database server does not automatically handle conversions between the **d_float** distinct type of **tab1** to the FLOAT type of table **tab2**:

```
INSERT INTO tab2 SELECT col1::ROW(a INT, b FLOAT) FROM tab1
```

In this example, the cast that is used to convert the **b** field is explicit because the conversion from **d_float** to FLOAT requires an explicit cast (to convert a distinct type to its source type requires an explicit cast).

In general, to cast between two unnamed row types where one or more of the fields uses an explicit cast, you must explicitly cast at the level of the row type, not at the level of the field.

Explicit casts on fields of a named row type

When you explicitly cast a value as a named row type, the database server automatically invokes any implicit or explicit casts that are used to convert field values to the target data type. In the following statement, the explicit cast of **col1** to type **row_t** automatically invokes the explicit cast that converts a field value of type FLOAT to **d_float**:

```
INSERT INTO tab3 SELECT col2::row_t FROM tab2
```

The following INSERT statement includes an explicit cast to the **row_t** type. The explicit cast to the row type also invokes an explicit cast to convert the **b** field of type **row_t** from FLOAT to **d_float**. In general, an explicit cast to a row type also invokes any explicit casts on the individual fields (one-level deep) that the row type contains to handle conversions.

```
INSERT INTO tab3 VALUES (ROW(5, 6.55::FLOAT)::row_t)
```

The following statement is also valid and returns the same results as the preceding statement. However, this statement shows all the explicit casts that are performed to insert a **row_t** value into the **tab3** table.

```
INSERT INTO tab3 VALUES (ROW(5, 6.55::float::d_float)::row_t)
```

In the preceding examples, the conversions between the **b** fields of the row types require two levels of casting. The database server handles any value that contains a decimal point as a DECIMAL type. In addition, no implicit casts exist between the DECIMAL and **d_float** data types, so two levels of casting are necessary: a cast from DECIMAL to FLOAT and a second cast from FLOAT to **d_float**.

Cast individual fields of a row type

If an operation on a field of a row type requires an explicit cast, you can explicitly cast the individual field value without consideration of the row type with which the field is associated. The following statement uses an explicit cast on the field value to handle the conversion:

```
SELECT col1 from tab1, tab2 WHERE col1.b = col2.b::FLOAT::d_float
```

If an operation on a field of a row type requires an implicit cast, you can specify the correct field value and the database server handles the conversion automatically. In the following statement, which compares field values of different data types, a built-in cast automatically converts between INT and FLOAT values:

```
SELECT col1 from tab1, tab2 WHERE col1.a = col2.b
```

Cast collection data types

In some cases, you can use an explicit cast to perform conversions between two collections with different element types. To compare or substitute between values of any two collection types, both collections must be of type SET, MULTISET, or LIST.

- Two element types are equivalent when all component types are the same. For example, if the element type of one collection is a row type, the other collection type is also a row type with the same number of fields and the same field data types.
- Casts exist in the database to perform conversions between any and all components of the element types that are not of the same data type.

  If the corresponding element types are not of the same data type, can use either built-in casts or user-defined casts to handle data conversions on the element types.

When the database server inserts, updates, or compares values of a collection data type, type checking occurs at the level of the element data type. Consequently, in a cast between two collection types, the data conversion occurs at the level of the element type because the actual data stored in a collection is of a particular element type.

The following type and tables are used in the collection casting examples in this section:

```
CREATE DISTINCT TYPE my_int AS INT;

CREATE TABLE set_tab1 (col1 SET(my_int NOT NULL));
CREATE TABLE set_tab2 (col2 SET(INT NOT NULL));
CREATE TABLE set_tab3 (col3 SET(FLOAT NOT NULL));
CREATE TABLE list_tab (col4 LIST(INT NOT NULL));
CREATE TABLE m_set_tab(col5 MULTISET(INT NOT NULL));
```

Restrictions on collection-type conversions

Because each collection data type (SET, MULTISET, and LIST) has different characteristics, conversions between collections with different collection types are disallowed. For example, elements stored in a LIST collection have a specific order associated with them. This order would be lost if the elements inserted into a LIST collection can be inserted into a MULTISET collection. Consequently, you cannot insert or update elements from one collection with elements from a different collection type even though the two collections might share the same element type. The following INSERT statement returns an error because the column on which the insert is performed is a MULTISET collection and the value being inserted is a LIST collection:

```
INSERT INTO m_set_tab SELECT col4 FROM list_tab -- returns error
```

Collections with different element types

How you handle conversions between two collections that have the same collection type but different element types depends on the element type of each collection and the type of cast that the database server uses to convert one element type to another when the element types are different, as follows:

- If a built-in cast or implicit user-defined cast exists to handle the conversion between two element types, you are not required to explicitly cast between the collection types.
- If an explicit cast exists to handle the conversion between element types, you can perform an explicit cast on a collection.

Implicit cast between element types

When an implicit cast exists in the database to convert between different element types of two collections, you are not required to use an explicit cast to insert or update elements from one collection into another collection. The following INSERT statement retrieves elements from the **set_tab2** table and inserts the elements into the **set_tab3** table. Although the collection column from **set_tab2** has an INT element type and the collection column from **set_tab3** has a FLOAT element type, a built-in cast implicitly handles the conversion between INT and FLOAT values. An explicit cast is unnecessary in this case.

```
INSERT INTO set_tab3 SELECT col2 FROM set_tab2
```

Explicit cast between element types

When a conversion between different element types of two collections is performed with an explicit cast, you must explicitly cast one collection to the other collection type. In the following example, the conversion between the element types (INT and **my_int**) requires an explicit cast. (A cast between a distinct type and its source type is always explicit).

The following INSERT statement retrieves elements from the **set_tab2** table and inserts the elements into the **set_tab1** table. The collection column from **set_tab2** has an INT element type, and the collection column from **set_tab1** has a **my_int** element type. Because the conversion between the element types (INT and **my_int**) requires an explicit cast, you must explicitly cast the collection type.

```
INSERT INTO set_tab1 SELECT col2::SET(my_int NOT NULL)
    FROM set_tab2
```

To perform an explicit cast on a collection type, you must include the constructor (SET, MULTISET, or LIST), the element type, and the NOT NULL keyword.

Convert relational data to a MULTISET collection

When you have data from a relational table you can use a collection subquery to cast a row value to a MULTISET collection. Suppose you create the following tables:

```
CREATE TABLE tab_a ( a_col INTEGER);
CREATE TABLE tab_b (ms_col MULTISET(ROW(a INT) NOT NULL) );
```

The following example shows how you might use a collection subquery to convert rows of INT values from the **tab_a** table to a MULTISET collection. All rows from **tab_a** are converted to a MULTISET collection and inserted into the **tab_b** table.

```
INSERT INTO tab_b VALUES (
    (MULTISET (SELECT a_col FROM tab_a)))
```

Cast distinct data types

A distinct type inherits none of the built-in casts of the built-in type that a distinct type might use as its source type. Consequently, the built-in casts that exist to implicitly convert a built-in data type to other data types are unavailable to the distinct type that uses the built-in type as its source type. However, when you create a distinct type on a built-in type, the database server provides two explicit casts to handle conversions from the distinct type to the built-in type and from the built-in type to the distinct type.

Explicit casts with distinct types

To compare or substitute between values of a distinct type and its source type, you must explicitly cast one type to the other. For example, to insert into or update a column of a distinct type with values of the source type, you must explicitly cast the values to the distinct type.

Suppose you create a distinct type, **int_type**, that is based on the INTEGER data type and a table with a column of type **int_type**, as follows:

```
CREATE DISTINCT TYPE int_type AS INTEGER;
CREATE TABLE tab_z(col1 int_type);
```

To insert a value into the **tab_z** table, you must explicitly cast the value for the **col1** column to **int_type**, as follows:

```
INSERT INTO tab_z VALUES (35::int_type)
```

Suppose you create a distinct type, **num_type**, that is based on the NUMERIC, data type and a table with a column of type **num_type**, as follows:

```
CREATE DISTINCT TYPE num_type AS NUMERIC;
CREATE TABLE tab_x (col1 num_type);
```

The distinct **num_type** inherits none of the system-defined casts that exist for the NUMERIC data type. Consequently, the following insert requires two levels of casting. The first cast converts the value 35 from INT to NUMERIC and the second cast converts from NUMERIC to **num_type**:

```
INSERT INTO tab_x VALUES (35::NUMERIC::num_type)
```

The following INSERT statement on the **tab_x** table returns an error because no cast exists to convert directly from an INT type to **num_type**:

```
INSERT INTO tab_x VALUES (70::num_type) -- returns error
```

Cast between a distinct type and its source type

Although data of a distinct type has the same representation as its source type, a distinct type cannot be compared directly to its source type. For this reason, when you create a distinct data type, automatically registers the following explicit casts:

- A cast from the distinct type to its source type
- A cast from the source type to the distinct type

Suppose you create two distinct types: one to handle movie titles and the other to handle music recordings. You might create the following distinct types that are based on the VARCHAR data type:

```
CREATE DISTINCT TYPE movie_type AS VARCHAR(30);
CREATE DISTINCT TYPE music_type AS VARCHAR(30);
```

You can then create the **entertainment** table that includes columns of type **movie_type**, **music_type**, and VARCHAR.

```
CREATE TABLE entertainment
(
video         movie_type,
compact_disc  music_type,
laser_disv    VARCHAR(30)
);
```

To compare a distinct type with its source type or vice versa, you must perform an explicit cast from one data type to the other. For example, suppose you want to check for movies that are available on both video and laser disc. The following statement requires an explicit cast in the WHERE clause to compare a value of a distinct type (**music_type**) with a value of its source type (VARCHAR). In this example, the source type is explicitly cast to the distinct type.

```
SELECT video FROM entertainment
   WHERE video = laser_disc::movie_type
```

However, you might also explicitly cast the distinct type to the source type as the following statement shows:

```
SELECT video FROM entertainment
   WHERE video::VARCHAR(30) = laser_disc
```

To perform a conversion between two distinct types that are defined on the same source type, you must make an intermediate cast back to the source type before casting to the target distinct type. The following statement compares a value of **music_type** with a value of **movie_type**:

```
SELECT video FROM entertainment
   WHERE video = compact_disc::VARCHAR(30)::movie_type
```

Add and drop casts on a distinct type

To enforce strong typing on a distinct type, the database server provides explicit casts to handle conversions between a distinct type and its source type. However, the creator of a distinct type can drop the existing explicit casts and create implicit casts, so that conversions between a distinct type and its source type do not require an explicit cast.

> ⚠️ **Important:** When you drop the explicit casts between a distinct type and its source type that the database server provides, and instead create implicit casts to handle conversions between these data types, you diminish the distinctiveness of the distinct type.

The following DROP CAST statements drop the two explicit casts that were automatically defined on the **movie_type**:

```
DROP CAST(movie_type AS VARCHAR(30));
DROP CAST(VARCHAR(30) AS movie_type);
```

After the existing casts are dropped, you can create two implicit casts to handle conversions between **movie_type** and VARCHAR. The following CREATE CAST statements create two implicit casts:

```
CREATE IMPLICIT CAST (movie_type AS VARCHAR(30));
CREATE IMPLICIT CAST (VARCHAR(30) AS movie_type);
```

You cannot create a cast to convert between two data types if such a cast already exists in the database.

If you create implicit casts to convert between the distinct type and its source type, you can compare the two types without an explicit cast. In the following statement, the comparison between the **video** column and the **laser_disc** column requires a conversion. Because an implicit cast has been created, the conversion between VARCHAR and **movie_type** is implicit.

```
SELECT video FROM entertainment
   WHERE video = laser_disc
```

Cast to smart large objects

The database server provides casts to allow the conversion of TEXT and BYTE objects to BLOB and CLOB data types. This feature allows users to migrate BYTE and TEXT data from legacy databases into BLOB and CLOB columns.

The following example shows how to use an explicit cast to convert a BYTE column value from the **catalog** table in the **stores_demo** database to a BLOB column value and update the **catalog** table in the **superstores_demo** database:

```
UPDATE catalog SET advert = ROW (
(SELECT cat_photo::BLOB FROM stores_demo:catalog
   WHERE catalog_num = 10027),
   advert.caption)
      WHERE catalog_num = 10027
```

The database server does not provide casts to convert BLOB to BYTE values or CLOB to TEXT values.

Create cast functions for user-defined casts

If your database contains opaque data types, distinct data types, or named row types, you might want to create user-defined casts that allow you to convert between the different data types. When you want to perform conversions between two data types that have the same storage structure, you can use the CREATE CAST statement without a cast function. However, in some cases you must create a cast function that you then register as a cast. You must create a cast function under the following conditions:

- The conversion is between two data types that have different storage structures
- The conversion involves the manipulation of values to ensure that data conversions are meaningful

The following sections show how to create and use user-defined casts that require cast functions.

An example of casting between named row types

Suppose you create the named row types and table shown in the next example. Although the named row types are structurally equivalent, **writer_t** and **editor_t** are unique data types.

```
CREATE ROW TYPE writer_t (name VARCHAR(30), depart CHAR(3));
CREATE ROW TYPE editor_t (name VARCHAR(30), depart CHAR(3));

CREATE TABLE projects
(
   book_title  VARCHAR(20),
   writer      writer_t,
```

```
    editor      editor_t
);
```

To handle conversions between two named row types, you must first create a user-defined cast. The following example creates a casting function and registers it as a cast to handle conversions from type **writer_t** to **editor_t**:

```
CREATE FUNCTION cast_rt (w writer_t)
   RETURNS editor_t
   RETURN (ROW(w.name, w.depart)::editor_t);
END FUNCTION;

CREATE CAST (writer_t as editor_t WITH cast_rt);
```

After you create and register the cast, you can explicitly cast values of type **writer_t** to **editor_t**. The following query uses an explicit cast in the WHERE clause to convert values of type **writer_t** to **editor_t**:

```
SELECT book_title FROM projects
   WHERE CAST(writer AS editor_t) = editor;
```

If you prefer, you can use the :: cast operator to perform the same cast, as the following example shows:

```
SELECT book_title FROM projects
   WHERE writer::editor_t = editor;
```

An example of casting between distinct data types

Suppose you want to define distinct types to represent dollar, yen, and sterling currencies. Any comparison between two currencies must take the exchange rate into account. Thus, you must create cast functions that not only handle the cast from one data type to the other data type but also calculate the exchange rate for the values that you want to compare.

The following example shows how you might define three distinct types on the same source type, DOUBLE PRECISION:

```
CREATE DISTINCT TYPE dollar AS DOUBLE PRECISION;
CREATE DISTINCT TYPE yen AS DOUBLE PRECISION;
CREATE DISTINCT TYPE sterling AS DOUBLE PRECISION;
```

After you define the distinct types, you can create a table that provides the prices that manufacturers charge for comparable products. The following example creates the **manufact_price** table, which contains a column for the dollar, yen, and sterling distinct types:

```
CREATE TABLE manufact_price
(
product_desc  VARCHAR(20),
us_price      dollar,
japan_price   yen,
uk_price      sterling
);
```

When you insert values into the **manufact_price** table, you can cast to the correct distinct type for dollar, yen, and sterling values, as follows:

```
INSERT INTO manufact_price
   VALUES ('baseball', 5.00::DOUBLE PRECISION::dollar,
   510.00::DOUBLE PRECISION::yen,
   3.50::DOUBLE PRECISION::sterling);
```

Because a distinct type does not inherit any of the built-in casts available to its source type, each of the preceding INSERT statements requires two casts. For each INSERT statement, the inner cast converts from DECIMAL to DOUBLE PRECISION and the outer cast converts from DOUBLE PRECISION to the correct distinct type (dollar, yen, or sterling).

Before you can compare the dollar, yen, and sterling data types, you must create cast functions and register them as casts. The following example creates SPL functions that you can use to compare dollar, yen, and sterling values. Each function multiplies the input value by a value that reflects the exchange rate.

```
CREATE FUNCTION dollar_to_yen(d dollar)
   RETURN (d::DOUBLE PRECISION * 106)::CHAR(20)::yen;
END FUNCTION;

CREATE FUNCTION sterling_to_dollar(s sterling)
   RETURNS dollar
   RETURN (s::DOUBLE PRECISION * 1.59)::CHAR(20)::dollar;
END FUNCTION;
```

After you write the cast functions, you must use the CREATE CAST statement to register the functions as casts. The following statements register the **dollar_to_yen()** and **sterling_to_dollar()** functions as explicit casts:

```
CREATE CAST(dollar AS yen WITH dollar_to_yen);
CREATE CAST(sterling AS dollar WITH sterling_to_dollar);
```

After you register the function as a cast, use it for operations that require conversions between the data types. For the syntax that you use to create a cast function and register it as a cast, see the CREATE FUNCTION and CREATE CAST statements in the *Informix® Guide to SQL: Syntax*.

In the following query, the WHERE clause includes an explicit cast that invokes the **dollar_to_yen()** function to compare dollar and yen values:

```
SELECT * FROM manufact_price
   WHERE CAST(us_price AS yen) < japan_price;
```

The following query uses the cast operator to perform the same conversion shown in the preceding query:

```
SELECT * FROM manufact_price
   WHERE us_price::yen < japan_price;
```

You can also use an explicit cast to convert the values that a query returns. The following query uses a cast to return yen equivalents of dollar values. The WHERE clause of the query also uses an explicit cast to compare dollar and yen values.

```
SELECT us_price::yen, japan_price FROM manufact_price
   WHERE us_price::yen < japan_price;
```

Multilevel casting

A *multilevel cast* refers to an operation that requires two or more levels of casting in an expression to convert a value of one data type to the target data type. Because no casts exist between yen and sterling values, a query that compares the two data types requires multiple casts. The first (inner) cast converts sterling values to dollar values; the second (outer) cast converts dollar values to yen values.

```
SELECT * FROM manufact_price
   WHERE japan_price < uk_price::dollar::yen
```

You might add another cast function to handle yen to sterling conversions directly. The following example creates the function **yen_to_sterling()** and registers it as a cast. To account for the exchange rate, the function multiplies yen values by .01 to derive equivalent sterling values.

```
CREATE FUNCTION yen_to_sterling(y yen)
   RETURNS sterling
   RETURN (y::DOUBLE PRECISION * .01)::CHAR(20)::sterling;
END FUNCTION;

CREATE CAST (yen AS sterling WITH yen_to_sterling);
```

With the addition of the yen to sterling cast, you can use a single-level cast to compare yen and sterling values, as the following query shows:

```
SELECT japan_price::sterling, uk_price FROM manufact_price
   WHERE japan_price::sterling) < uk_price;
```

In the SELECT statement, the explicit cast returns yen values as their sterling equivalents. In the WHERE clause, the cast allows comparisons between yen and sterling values.

Index