Informix® Guide to Database Design and Implementation

Informix Dynamic Server, Version 7.3
Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.2

Informix Dynamic Server, Developer Edition, Version 7.3
Informix Dynamic Server, Workgroup Edition, Version 7.3

February 1998
Part No. 000-4364

Published by INFORMIX® Press

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

Documentation Team: Twila Booth, Brian Deutscher, Evelyn Eldridge-Diaz

Table of Contents

Introduction

About This Manual 3
Types of Users 4
Software Dependencies 4
Assumptions About Your Locale 5
Demonstration Databases 5
New Features 6
New Features in Version 7.3 6
New Features in Version 8.2 6
Documentation Conventions 7
Typographical Conventions 8
Icon Conventions 9
Sample-Code Conventions 11
Additional Documentation 12
On-Line Manuals 12
Printed Manuals 13
Error Message Files 13
Documentation Notes, Release Notes, Machine Notes 14
Related Reading 15
Compliance with Industry Standards 16
Informix Welcomes Your Comments 16

Section I Basics of Database Design and Implementation

Chapter 1 Planning a Database

Choosing a Data Model for Your Database 1-3
Using ANSI-Compliant Databases 1-4
Designating a Database as ANSI Compliant 1-5
Determining If an Existing Database Is ANSI Compliant 1-5
Differences Between ANSI-Compliant and Non-ANSI-Compliant Databases 1-6
Using a Customized Language Environment for Your Database 1-10

Chapter 2 Building a Relational Data Model

Why Build a Data Model 2-3
Overview of Entity-Relationship Data Model 2-4
Identifying and Defining Principal Data Objects 2-5
Discovering Entities 2-5
Defining the Relationships 2-9
Identifying Attributes 2-17
Diagramming Data Objects 2-19
Reading E-R Diagrams 2-20
The Telephone-Directory Example 2-21
Translating E-R Data Objects into Relational Constructs 2-22
Defining Tables, Rows, and Columns 2-23
Determining Keys for Tables 2-25
Resolving Relationships 2-29
Resolving m:n Relationships 2-29
Resolving Other Special Relationships 2-30
Normalizing a Data Model 2-31
First Normal Form 2-32
Second Normal Form 2-34
Third Normal Form 2-34
Summary of Normalization Rules 2-35

Chapter 3 Choosing Data Types

Defining the Domains 3-3
Data Types 3-4
Null Values 3-23
Default Values 3-23
Check Constraints 3-24

Chapter 4 Implementing a Relational Data Model

Creating the Database 4-3
Using CREATE DATABASE 4-4
Using CREATE TABLE 4-6
Using Synonyms with Table Names 4-8
Using Synonym Chains 4-10
Using Command Scripts 4-11
Populating the Tables 4-12

Chapter 5 Fragmentation Strategies

What Is Fragmentation? 5-3
Enhanced Fragmentation for Dynamic Server with AD and XP Options 5-6
Why Use Fragmentation? 5-6
Whose Responsibility Is Fragmentation? 5-8
Fragmentation and Logging 5-8
Distribution Schemes for Table Fragmentation 5-9
Round-Robin Distribution Scheme 5-11
Expression-Based Distribution Schemes 5-11
System-Defined Hash Distribution Scheme 5-13
Hybrid Distribution Scheme 5-14
When Can the Database Server Eliminate Fragments from a Search? 5-15
Creating a Fragmented Table 5-26
Creating a New Fragmented Table 5-27
Creating a Fragmented Table from Nonfragmented Tables 5-28
Modifying a Fragmented Table 5-30
Modifying Fragmentation Strategies 5-30
Fragmenting Temporary Tables 5-37
Fragmenting Temporary Tables for Dynamic Server with AD and XP Options 5-37
Fragmentation of Table Indexes 5-39
Attached Indexes 5-39
Detached Indexes 5-40
Rowids 5-41
Accessing Data Stored in Fragmented Tables 5-42
Using Primary Keys Instead of Rowids 5-42

Section II Data Warehousing

Chapter 6 Building a Dimensional Data Model

Overview of Data Warehousing 6-4
Why Build a Dimensional Database? 6-5
What is Dimensional Data? 6-6
Concepts of Dimensional Data Modeling 6-9
The Fact Table 6-10
Dimensions of the Data Model 6-11
Building a Dimensional Data Model 6-15
Choosing a Business Process 6-16
Summary of a Business Process 6-16
Determining the Granularity of the Fact Table 6-18
Identifying the Dimensions and Hierarchies 6-20
Choosing the Measures for the Fact Table 6-22
Resisting Normalization 6-25
Choosing the Attributes for the Dimension Tables 6-26
Handling Common Dimensional Data-Modeling Problems . . . 6-28
Minimizing the Number of Attributes in a Dimension Table 6-28
Handling Dimensions That Occasionally Change 6-30
Using the Snowflake Schema. 6-32

Chapter 7 Implementing a Dimensional Data Model

Implementing the Dimensional Database 7-3
Using CREATE DATABASE 7-3
Using CREATE TABLE for the Dimension and Fact Tables 7-4
Mapping Data from Data Sources to the Database 7-6
Loading Data into the Dimensional Database 7-9
Using Command Files to Create the sales_demo Database 7-11
Testing the Dimensional Database 7-12
Logging and Nonlogging Tables for Dynamic Server with AD and XP Options 7-13
Choosing Table Types 7-14
Switching Between Table Types 7-18
Indexes for Data-Warehousing Environments 7-18
Using GK Indexes in a Data-Warehousing Environment . . . 7-20

Section III Managing Databases

Chapter 8 Granting and Limiting Access to Your Database

Controlling Access to Databases 8-4
Securing Confidential Data 8-4
Granting Privileges 8-5
Database-Level Privileges 8-5
Ownership Rights 8-7
Table-Level Privilege 8-8
Column-Level Privileges 8-10
Procedure-Level Privileges 8-12
Automating Privileges 8-13
Using Stored Procedures to Control Access to Data 8-17
Restricting Data Reads 8-18
Restricting Changes to Data 8-19
Monitoring Changes to Data 8-20
Restricting Object Creation 8-21
Using Views 8-22
Creating Views 8-23
Modifying with a View 8-26
Privileges and Views 8-30
Privileges When Creating a View 8-30
Privileges When Using a View 8-31

Index

Introduction

About This Manual 3
Types of Users 4
Software Dependencies 4
Assumptions About Your Locale 5
Demonstration Databases 5

New Features 6
New Features in Version 7.3 6
New Features in Version 8.2 6

Documentation Conventions 7
Typographical Conventions 8
Icon Conventions 9
Comment Icons 9
Feature, Product, and Platform Icons 10
Compliance Icons 11
Sample-Code Conventions 11

Additional Documentation 12
On-Line Manuals 12
Printed Manuals 13
Error Message Files 13
Documentation Notes, Release Notes, Machine Notes 14
Related Reading 15

Compliance with Industry Standards 16

Informix Welcomes Your Comments 16

Read this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

This manual 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 manual is one of several manuals that discuss the Informix implementation of Structured Query Language (SQL). This manual shows how to use SQL data definition statements to implement your databases. The *Informix Guide to SQL: Tutorial* shows how to use basic and advanced SQL to access and manipulate the data in your databases. The *Informix Guide to SQL: Syntax* contains all the syntax descriptions for SQL and stored procedure language (SPL). The *Informix Guide to SQL: Reference* provides reference information for aspects of SQL other than the language statements.

Types of Users

This manual is for the following users:

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

This manual 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

If you have limited experience with relational databases, SQL, or your operating system, refer to the *Getting Started* manual for your database server for a list of supplementary titles.

Software Dependencies

This manual assumes that you are using one of the following database servers:

- Informix Dynamic Server, Version 7.3
- Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.2
- Informix Dynamic Server, Developer Edition, Version 7.3
- Informix Dynamic Server, Workgroup Edition, Version 7.3

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a GLS (Global Language Support) locale.

This manual assumes that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

Demonstration Databases

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. You can use SQL scripts provided with DB-Access to derive a second database, called **sales_demo**. This database illustrates a dimensional schema for data-warehousing applications. Sample command files are also included for creating and populating these databases.

Many examples in Informix manuals are based on the **stores7** demonstration database. The **stores7** database is described in detail and its contents are listed in the *Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and the **%INFORMIXDIR%\bin** directory on Windows NT platforms. For a complete explanation of how to create and populate the **stores7** demonstration database, refer to the *DB-Access User Manual*. For an explanation of how to create and populate the **sales_demo** database, see Chapter 7, "Implementing a Dimensional Data Model."

New Features

The following sections describe new database server features relevant to this manual. For a comprehensive list of new features, see the release notes for your database server.

New Features in Version 7.3

Most of the new features for Version 7.3 of Informix Dynamic Server fall into five major areas:

- Reliability, availability, and serviceability
- Performance
- Windows NT-specific features
- Application migration
- Manageability

Several additional features affect connectivity, replication, and the optical subsystem.

This manual describes enhancements to the CREATE VIEW statement, a feature that is related to application migration.

New Features in Version 8.2

This manual provides information about the following new features that were implemented in Version 8.2 of Dynamic Server with AD and XP Options:

- Enhanced fragmentation of table data on multiple computers
- Enhanced indexes to support data-warehousing applications
- Global Language Support (GLS)

This manual also discusses the following features, which were introduced in Version 8.1 of Dynamic Server with AD and XP Options:

- Fragmentation of table data across multiple computers
- New join methods for use across multiple computers
- Nonlogging tables
- External tables for high-performance loading and unloading
- Dbslices for centralized administration of storage spaces
- Enhancements to the CREATE VIEW statement

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Sample-code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information within a table or section.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after you type the indicated information on your keyboard. When you are instructed to "type" the text or to "press" other keys, you do not need to press* RETURN.

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
	The *warning* icon identifies vital instructions, cautions, or critical information.
	The *important* icon identifies significant information about the feature or operation that is being described.
	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
AD/XP	Identifies information that is specific to Dynamic Server with AD and XP Options.
E/C	Identifies information that is specific to the INFORMIX-ESQL / C product.
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature.
IDS	Identifies information that is specific to Dynamic Server and its editions. However, in some cases, the identified section applies only to Informix Dynamic Server and not to Informix Dynamic Server, Workgroup and Developer Editions. Such information is clearly identified.
UNIX	Identifies information that is specific to UNIX platforms.
W/D	Identifies information that is specific to Informix Dynamic Server, Workgroup and Developer Editions.
WIN NT	Identifies information that is specific to the Windows NT environment.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of the feature-, product-, or platform-specific information that appears within a table or a set of paragraphs within a section.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database.
+	Identifies information that is an Informix extension to ANSI SQL-92 entry-level standard SQL.
X/O	Identifies functionality that conforms to X/Open.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. If an icon appears next to a section heading, the compliance information ends at the next heading at the same or higher level. A ♦ symbol indicates the end of compliance information that appears in a table row or a set of paragraphs within a section.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores7
.
.
.
DELETE FROM customer
    WHERE customer_num = 121
.
.
.
COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes, release notes, and machine notes
- Related reading

On-Line Manuals

An Answers OnLine CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine.

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com. Please provide the following information when you place your order:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. For a detailed description of these error messages, refer to *Informix Error Messages* in Answers OnLine.

UNIX

To read the error messages in UNIX, you can use the following commands.

Command	Description
finderr	Displays error messages on line
rofferr	Formats error messages for printing

♦

WIN NT

To read error messages and corrective actions in Windows NT, use the **Informix Find Error** utility. To display this utility, choose **Start→Programs→Informix** from the Task Bar. ♦

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following sections describe the on-line files that supplement the information in this manual. Please examine these files before you begin using your database server. They contain vital information about application and performance issues.

UNIX

On UNIX platforms, the following on-line files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

On-Line File	Purpose
DDIDOC_*x.y*	The documentation-notes file for your version of this manual describes features that are not covered in the manual or that have been modified since publication. Replace *x.y* in the filename with the version number of your database server to derive the name of the documentation-notes file for this manual.
SERVERS_*x.y*	The release-notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds. Replace *x.y* in the filename with the version number of your database server to derive the name of the release-notes file.
IDS_*x.y*	The machine-notes file describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product they describe. Replace *x.y* in the filename with the version number of your database server to derive the name of the machine-notes file.

◆

WIN NT

The following items appear in the Informix folder. To display this folder, choose **Start→Programs→Informix** from the Task Bar.

Item	Description
Documentation Notes	This item includes additions or corrections to manuals, along with information about features that may not be covered in the manuals or that have been modified since publication.
Release Notes	This item describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Machine notes do not apply to Windows NT platforms. ♦

Related Reading

The following publications provide additional information about the topics that are discussed in this manual. For a list of publications that provide an introduction to database servers and operating-system platforms, refer to the *Getting Started* manual.

If you are interested in learning more about fundamental concepts and approaches to database design, Informix recommends the following books:

- *Database Modeling and Design*, *The Entity-Relationship Approach* by Toby J. Teorey (Morgan Kauffman Publishers, Inc., 1990)
- *Handbook of Relational Database Design* by Candace C. Fleming and Barbara von Halle (Addison-Wesley Publishing Company, 1989)

If you are interested in database design for data warehousing, Informix recommends the following books:

- *The Data Warehouse Toolkit* by Ralph Kimball
  (John Wiley & Sons, Inc., 1996)
- *Building the Data Warehouse* by W.H. Inmon
  (John Wiley & Sons, Inc., 1996)

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

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

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

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send email, our address is:

doc@informix.com

Or send a facsimile to the Informix Technical Publications Department at:

650-926-6571

We appreciate your feedback.

Basics of Database Design and Implementation

Section I

Chapter

1

Planning a Database

Choosing a Data Model for Your Database 1-3

Using ANSI-Compliant Databases 1-4
- Designating a Database as ANSI Compliant 1-5
- Determining If an Existing Database Is ANSI Compliant 1-5
- Differences Between ANSI-Compliant and Non-ANSI-Compliant Databases 1-6
  - Transactions 1-6
  - Transaction Logging. 1-7
  - Owner Naming 1-7
  - Privileges on Objects 1-7
  - Default Isolation Level 1-8
  - Character Data Types 1-8
  - Decimal Data Type 1-8
  - Escape Characters 1-9
  - Cursor Behavior 1-9
  - The SQLCODE Field of the SQL Communications Area . . . 1-9
  - SQL Statements Allowed with ANSI-Compliant Databases. 1-10
  - Synonym Behavior 1-10

Using a Customized Language Environment for Your Database . . . 1-10

This chapter describes several issues that a database administrator (DBA) must understand to effectively plan for a database. It discusses the following topics:

- Choosing a Data Model for Your Database
- Using ANSI-Compliant Databases
- Using a Customized Language Environment for Your Database

Choosing a Data Model for Your Database

Before you create a database with an Informix product, you must decide what type of data model you want to use to design your database. This manual describes two data models that you can use to design your database.

The first model is a traditional relational data model that typifies database design for on-line transaction processing (OLTP). The purpose of transaction processing is to process a large number of very small, atomic transactions without losing any of them. An OLTP database is designed to handle the day-to-day operational needs of the business and database performance is tuned for those operational needs. Section I of this manual, "Basics of Database Design and Implementation" describes how to build and implement a relational data model for OLTP.

The second model is a dimensional data model that shows basic database design for data warehousing. In a data warehousing environment, databases are optimized for data retrieval and analysis. This type of informational processing is known as on-line analytical processing (OLAP) or decision support processing. Section II, "Data Warehousing" describes how to build and implement a dimensional data model for OLAP.

In addition to the data model you choose to design the database, you must make the following decisions that determine which features are available to applications that use the database:

- Which database server houses the database: Informix Dynamic Server, Informix Dynamic Server, Workgroup and Developer Editions, or Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options?
- Does the database need to be ANSI compliant?
- Will the database use characters from a language other than English in its tables?

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

Using ANSI-Compliant Databases

You create an ANSI-compliant database when you use the MODE ANSI keywords in the CREATE DATABASE statement. The differences between ANSI-compliant databases and those that are not ANSI-compliant are described on page 1-6.

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. However, creating an ANSI-compliant database does not ensure that this database remains compliant with the ANSI/ISO SQL-92 standards. (If you take a non-ANSI action, such as CREATE INDEX, on an ANSI database, you receive a warning, but the application program does not forbid the action.)

- Name isolation

  The ANSI table-naming scheme allows different users to create tables in a database without having to worry about name conflicts.

- Transaction isolation
- Data recovery

  ANSI-compliant databases enforce unbuffered logging and automatic transactions for Dynamic Server.

Designating a Database as ANSI Compliant

Once you create a database with the MODE ANSI keywords, the database is considered ANSI compliant. In an ANSI-compliant database, you cannot change the logging mode to buffered logging, and you cannot turn logging off.

Determining If an Existing Database Is ANSI Compliant

The following list describes several methods to determine if a database is ANSI compliant:

- If the default database server is Dynamic Server, you can use the ON-Monitor utility to list all the databases that reside on that default database server. The Databases option of the Status menu displays this list. In the Log Status column on the list, an ANSI-compliant database has the notation `U*`.
- If the default database server is Dynamic Server with AD and XP Options, you can use the Informix Enterprise Command Center (IECC) to list all the databases that reside on that default 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 on SQLCA, see the *Informix Guide to SQL: Tutorial* or your SQL API manual.

Differences Between ANSI-Compliant and Non-ANSI-Compliant Databases

Databases that you designate as ANSI compliant (by using the MODE ANSI keywords when you create them) and databases that are not ANSI compliant behave differently in the following areas:

- Transactions
- Transaction logging
- Owner naming
- Privileges on objects
- Default isolation level
- Character data types
- Decimal data type
- Escape characters
- Cursor behavior
- SQLCODE of the SQLCA
- Allowed SQL statements
- Synonym behavior

Transactions

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, you can choose whether to use transaction processing.

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 redundant and unnecessary because all statements are automatically contained in a transaction. You need to indicate only the end of a transaction with a COMMIT WORK or ROLLBACK WORK statement.

For more information on transactions, see the *Informix Guide to SQL: Tutorial* and Chapter 4, "Implementing a Relational Data Model" in this manual.

Transaction Logging

All ANSI-compliant databases on Dynamic Server and Dynamic Server with AD and XP Options run with unbuffered transaction logging.

IDS

Databases 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. ♦

AD/XP

Databases that are not ANSI compliant run with unbuffered logging only. Unbuffered logging provides more comprehensive data recovery. ♦

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 databases that are not ANSI compliant 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 (as well as 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, Informix provides two table-level privileges, Alter and Index, that are not included in the ANSI standards.

For more information about how to grant table-level privileges, see Chapter 8, "Granting and Limiting Access to Your Database" in this manual and the description of the GRANT statement in the *Informix Guide to SQL: Syntax*.

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

For more information on stored procedure privileges, see the *Informix Guide to SQL: Tutorial*.

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 databases that are not ANSI compliant but do use logging is Committed Read on Dynamic Server and Dynamic Server with AD and XP Options. The default isolation level for databases that are not ANSI compliant and do not use logging is Uncommitted Read.

For information on 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

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

Decimal Data Type

In an ANSI-compliant database, no scale is used for the DECIMAL data type. You can think of this as scale = 0.

Escape Characters

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

Cursor Behavior

If a database is not ANSI compliant, you need to 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.

With an ANSI-compliant database, you do not have to explicitly use the FOR UPDATE keywords when you declare a cursor. In ANSI-compliant databases, *all* cursors that meet the restrictions described in the preceding list 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*.

The 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 sets SQLCODE to `0` if the database is not ANSI compliant.

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

SQL Statements Allowed with ANSI-Compliant Databases

No restrictions exist on SQL statements that are allowed in applications that you use with an ANSI-compliant database. You can use the Informix extensions with either an ANSI-compliant database or a database that is not ANSI compliant.

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.

GLS

Using a Customized Language Environment for Your Database

With Global Language Support (GLS), Informix Version 7.2 and later products permit you to use different locales. A GLS locale is an environment that has defined conventions for a particular language or culture.

Tip: *With Version 7.2 and later products, GLS replaces Native Language Support (NLS) and Asian Language Support (ALS).*

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 functionalities:

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

For descriptions of GLS environment variables and for detailed information on how to implement non-U.S. English environments, see the *Informix Guide to GLS Functionality*.

Chapter

2

Building a Relational Data Model

Why Build a Data Model 2-3

Overview of Entity-Relationship Data Model 2-4

Identifying and Defining Principal Data Objects 2-5
- Discovering Entities 2-5
  - Choosing Possible Entities 2-5
  - The List of Entities 2-6
  - Telephone-Directory Example 2-7
  - Diagramming Entities 2-9
- Defining the Relationships 2-9
  - Connectivity 2-10
  - Existence Dependency 2-10
  - Cardinality 2-11
  - Discovering the Relationships 2-11
  - Diagramming Relationships 2-16
- Identifying Attributes 2-17
  - Selecting Attributes for Entities 2-17
  - Listing Attributes. 2-18
  - About Entity Occurrences 2-18

Diagramming Data Objects 2-19
- Reading E-R Diagrams 2-20
- The Telephone-Directory Example 2-21

Translating E-R Data Objects into Relational Constructs 2-22
- Defining Tables, Rows, and Columns 2-23
  - Placing Constraints on Columns 2-24
- Determining Keys for Tables 2-25
  - Primary Keys 2-25
  - Foreign Keys (Join Columns). 2-27
  - Adding Keys to the Telephone-Directory Diagram 2-28

Resolving Relationships 2-29
Resolving m:n Relationships 2-29
Resolving Other Special Relationships 2-30

Normalizing a Data Model 2-31
First Normal Form 2-32
Second Normal Form 2-34
Third Normal Form 2-34
Summary of Normalization Rules 2-35

The first step in creating a traditional relational database is to construct a data model: a precise, complete definition of the data you want to store. This chapter provides a cursory overview of one way to model the data. Chapter 3, "Choosing Data Types" discusses the column-specific properties that you define to complete a data model. Chapter 4, "Implementing a Relational Data Model" describes how to implement the data model described in this chapter.

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

Why Build a Data Model

You already have some idea about the type of data in your database and how that data needs to be organized. This information is the beginning of a data model. When you use some type of formal notation to build your data model, you can help your design in two ways:

- You think through the data model completely.

  A mental model often contains unexamined assumptions; when you formalize the design, you discover these assumptions.

- Your 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 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, a modeling method taught in Informix training courses. The E-R data modeling method uses the following steps:

1. Identify and define the principal data objects (entities, relationships, and attributes).
2. Use the E-R approach to diagram the data objects.
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 discussed in this chapter. Chapter 4, "Implementing a Relational Data Model" discusses 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 2-21 on page 2-33, 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.

Identifying and Defining Principal Data Objects

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

Discovering 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 its nouns. The demonstration database contains the following entities: *customer*, *orders*, *items*, *stock*, *catalog*, *cust_calls*, *call_type*, *manufact*, and *state*.

Choosing Possible Entities

You can probably list several entities for your database immediately. Make a preliminary list of all the entities you can identify. Interview the potential users of the database for their opinions about what must be recorded in the database. Determine basic characteristics for each entity, such as "at least one address must be associated with a name." All the decisions you make about the entities become your *business rules*. "Telephone-Directory Example" on page 2-7 provides some of the business rules for the example in this chapter.

Later, when you *normalize* your data model, some of the entities can expand or become other data objects. For more information, see "Normalizing a Data Model" on page 2-31.

The 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, that do not need something 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 model (see "Telephone-Directory Example" on page 2-7), 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 further 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 needs.

First define the entities. Look carefully at a page from a telephone directory to identify the entities that it contains. Figure 2-1 shows a sample page from a telephone directory.

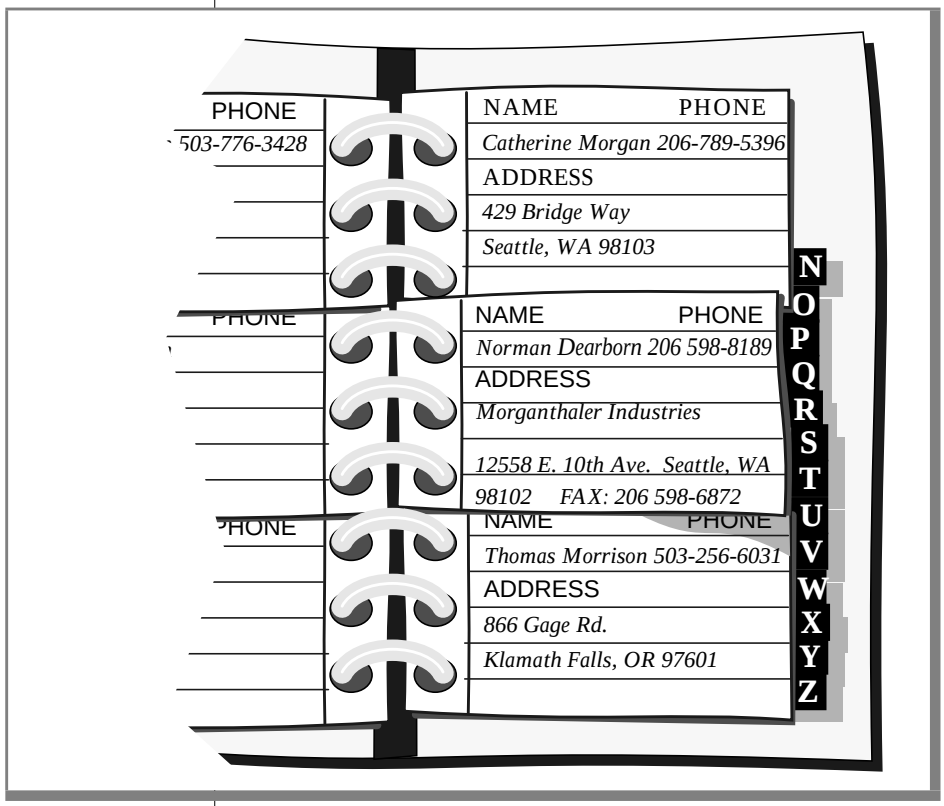


Figure 2-1
Partial Page from a Telephone Directory

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.

Are the Entities Generic and Significant?

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.

Are the Entities Fundamental?

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 are 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.

Are the Entities Unitary?

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 only one kind of address exists; you do not need 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)

Diagramming 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 Figure 2-2 shows. "Diagramming Data Objects" on page 2-19 shows how to put the entities together with relationships.

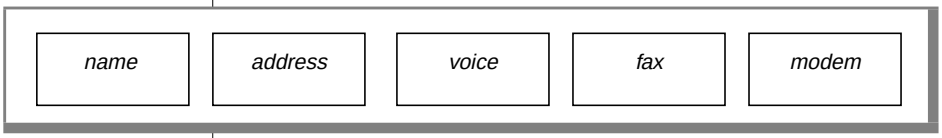


Figure 2-2 *Entities in the Personal Telephone-Directory Example*

Defining the Relationships

After you choose your database entities, you need to consider the relationships between them. Relationships are not always obvious, but all the ones worth recording must be found. The only way to ensure that all the relationships are found is to list all possible relationships exhaustively. Consider every pair of entities *A* and *B* and ask, "What is the relationship between an *A* and a *B*?"

A relationship is an association between two entities. Usually, a verb or preposition that connects two entities implies a relationship. A relationship between entities is described in terms of *connectivity*, *existence dependency*, and *cardinality*.

Connectivity

Connectivity refers to the number of entity instances. An entity instance is a particular occurrence of an entity. Figure 2-3 shows the three types of connectivity are one-to-one (written 1:1), one-to-many (written 1:n), and many-to-many (written m:n).

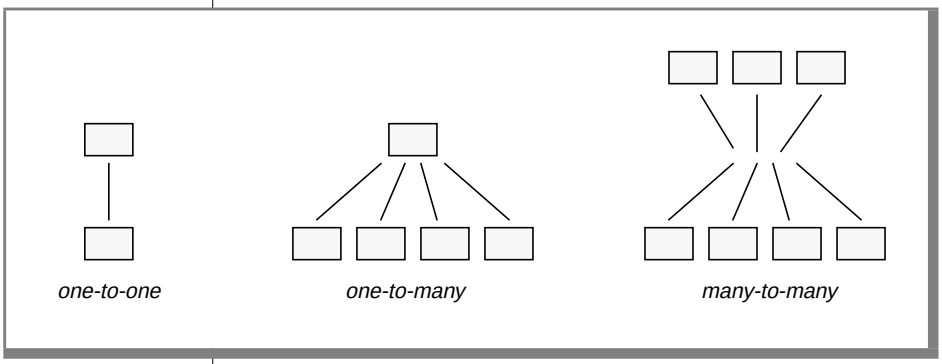


Figure 2-3
Connectivity in Relationships

For instance, in the telephone-directory example, an address can be associated with more than one name. The connectivity for the relationship between the name and address entities is one-to-many (1:n).

Existence Dependency

Existence dependency describes whether an entity in a relationship is optional or mandatory. Analyze your business rules to identify whether an entity must exist in a relationship. For example, your business rules might dictate that an address must be associated with a name. Such an association indicates a mandatory existence dependency for the relationship between the name and address entities. An example of an optional existence dependency could be a business rule that says a person might or might not have children.

Cardinality

Cardinality places a constraint on the number of times an entity can appear in a relationship. The cardinality of a 1:1 relationship is always one. But the cardinality of a 1:n relationship is open; *n* could be any number. If you need to place an upper limit on *n*, you specify a cardinality for the relationship. For instance, in a store sale example, you could limit the number of sale items that a customer can purchase at one time. You usually use your application program or a stored procedure to place cardinality constraints.

For more information about cardinality, see any textbook about E-R data modeling.

Discovering the Relationships

A convenient way to discover the relationships is to prepare a matrix that names all the entities on the rows and again on the columns. The matrix in Figure 2-4 reflects the entities for the personal telephone directory.

	name	address	number (voice)	number (fax)	number (modem)
name					
address					
number (voice)					
number (fax)					
number (modem)					

Figure 2-4 *A Matrix That Reflects the Entities for a Personal Telephone Directory*

You can ignore the shaded portion of the matrix. You must consider the diagonal cells; that is, you must ask the question "What is the relationship between an *A* and another *A*?" In this model, the answer is always none. No relationship exists between a name and a name or an address and another address, at least none that you need to record in this model. When a relationship exists between an *A* and another *A*, you have found a *recursive* relationship. (See "Resolving Other Special Relationships" on page 2-30.)

For all cells for which the answer is clearly none, write none in the matrix. Figure 2-5 shows the current matrix.

	name	address	number (voice)	number (fax)	number (modem)
name	none				
address		none			
number (voice)			none		
number (fax)				none	
number (modem)					none

Figure 2-5
A Matrix with Initial Relationships Included

Although no entities relate to themselves in this model, this is not always true in other models. A typical example is an employee who is the manager of another employee. Another example occurs in manufacturing, when a part entity is a component of another part.

In the remaining cells, you write the connectivity relationship that exists between the entity on the row and the entity on the column. The following kinds of relationships are possible:

- *One-to-one* (1:1), in which never more than one entity *A* exists for one entity *B* and never more than one *B* for one *A*.
- *One-to-many* (1:n), in which more than one entity *A* never exists, but several entities *B* can be related to *A* (or vice versa).
- *Many-to-many* (m:n), in which several entities *A* can be related to one *B* and several entities *B* can be related to one *A*.

One-to-many relationships are the most common. The telephone-directory model examples show one-to-many and many-to-many relationships.

As Figure 2-5 on page 2-12 shows, the first unfilled cell represents the relationship between names and addresses. What connectivity lies between these entities? You might ask yourself, "How many names can be associated with an address?" You decide that a name can have *zero* or *one* address but no more than one. You write `0-1` opposite **name** and below **address**, as Figure 2-6 shows.

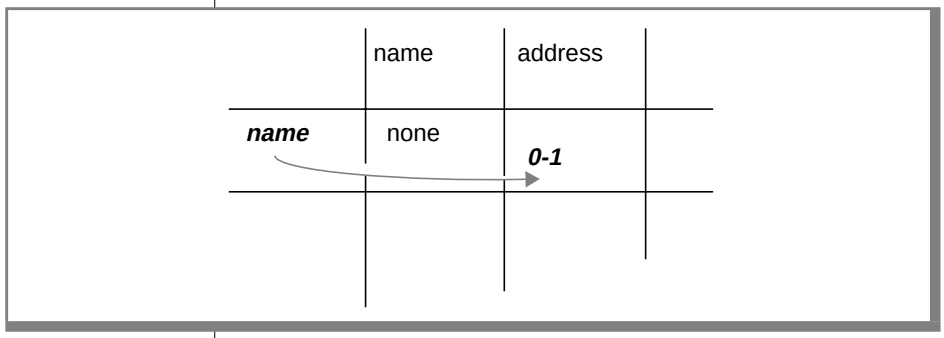


Figure 2-6
Relationship Between Name and Address

Ask yourself how many addresses can be associated with a name. You decide that an address can be associated with more than one name. For example, you can know several people at one company or more than two people who live at the same address.

Can an address be associated with *zero* names? That is, should it be possible for an address to exist when no names use it? You decide that yes, it can. Below **address** and opposite **name**, you write `0-n`, as Figure 2-7 shows.

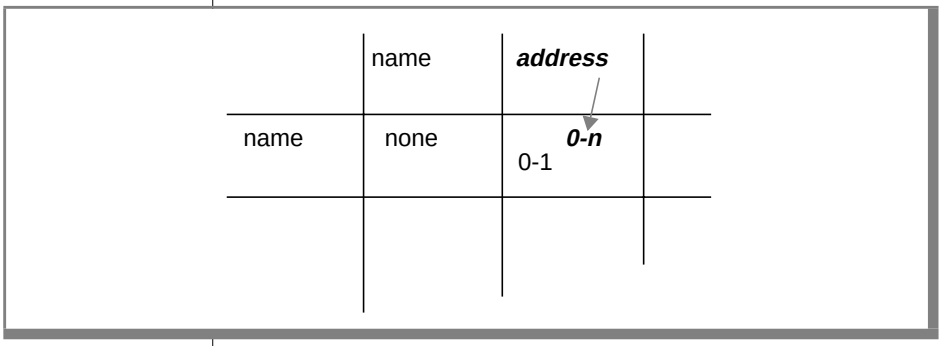


Figure 2-7
Relationship Between Address and Name

If you decide that an address cannot exist unless it is associated with at least one name, you write `1-n` instead of `0-n`.

When the cardinality of a relationship is limited on either side to `1`, it is a 1:n relationship. In this case, the relationship between names and addresses is a 1:n relationship.

Now consider the next cell in Figure 2-5 on page 2-12: the relationship between a name and a voice number. How many voice numbers can a name be associated with, one or more than one? When you look at your telephone directory, you see that you have often noted more than one telephone number for a person. For a busy salesperson you have a home number, an office number, a paging number, and a car phone number. But you might also have names without associated numbers. You write `0-n` opposite **name** and below **number (voice)**, as Figure 2-8 shows.

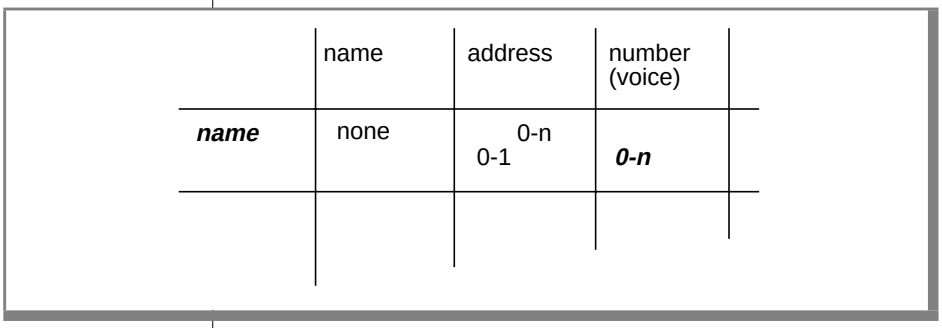


Figure 2-8
Relationship Between Name and Number

What is the other side of this relationship? How many names can be associated with a voice number? You decide that only one name can be associated with a voice number. Can a number be associated with zero names? No, you decide you do not need to record a number unless someone uses it. You write 1 under **number (voice)** and opposite **name**, as Figure 2-9 shows.

	name	address	***number (voice)***
name	none	0-n 0-1	***1*** 0-n

Figure 2-9 *Relationship Between Number and Name*

Take the following factors into account, to fill out the rest of the matrix in the same fashion:

- A name can be associated with more than one fax number; for example, a company can have several fax machines. Conversely, a fax number can be associated with more than one name; for example, several people can use the same fax number.
- A modem number must be associated with exactly one name. (This is an arbitrary decree to complicate the example; pretend it is a requirement of the design.) However, a name can have more than one associated modem number; for example, a company computer can have several dial-up lines.
- Although some relationship exists between a voice number and an address, a modem number and an address, and a fax number and an address in the real world, none needs to be recorded in this model. An indirect relationship already exists through *name*.

Figure 2-10 shows a completed matrix.

	name	address	number (voice)	number (fax)	number (modem)
name	none	0-n 0-1	1 0-n	1-n 0-n	1 0-n
address		none	none	none	none
number (voice)			none	none	none
number (fax)				none	none
number (modem)					none

Figure 2-10
A Completed Matrix for a Telephone Directory

Other decisions that the matrix reveals are that no relationship exists between a fax number and a modem number, between a voice number and a fax number, or between a voice number and a modem number.

You might disagree with some of these decisions (for example, that a relationship between voice numbers and modem numbers is not supported). For the sake of this example, these are our business rules.

Diagramming Relationships

For now, save the matrix that you created in this section. You will learn how to create an E-R diagram in "Diagramming Data Objects" on page 2-19.

Identifying Attributes

Entities contain *attributes*, which are characteristics or modifiers, qualities, amounts, or features. An attribute is a fact or nondecomposable piece of information about an entity. Later, when you represent an entity as a table, its attributes are added to the model as new columns.

You must identify the entities before you can identify the database attributes. After you determine the entities, ask yourself, "What characteristics do I need to know about each entity?" For example, in an *address* entity, you probably need information about *street*, *city*, and *zipcode*. Each of these characteristics of the *address* entity becomes an attribute.

Selecting Attributes for Entities

To select attributes, choose ones that have the following qualities:

- They are significant.

  Include only attributes that are useful to the database users.

- They are direct, not derived.

  An attribute that can be derived from existing attributes (for instance, through an expression in a SELECT statement) should not be part of the model. Derived data complicates the maintenance of a database.

  At a later stage of the design, you can consider adding derived attributes to improve performance, but at this stage exclude them. For information about how to improve the performance of your database server, see your *Performance Guide*.

- They are nondecomposable.

  An attribute can contain only single values, never lists or repeating groups. Composite values must be separated into individual attributes.

- They contain data of the same type.

  For example, you would want to enter only date values in a birthday attribute, not names or telephone numbers.

The rules for how to define attributes are the same as those for how to find columns. For information about how to define columns, see "Placing Constraints on Columns" on page 2-24.

The following attributes are added to the telephone-directory example to produce the diagram shown in Figure 2-15 on page 2-21:

- Street, city, state, and zip code are added to the *address* entity.
- Birth date, e-mail address, anniversary date, and children's first names are added to the *name* entity.
- Type is added to the *voice* entity to distinguish car phones, home phones, and office phones. A voice number can be associated with only one voice type.
- The hours that a fax machine is attended are added to the *fax* entity.
- Whether a modem supports 9,600-, 14,400-, or 28,800-baud rates is added to the *modem* entity.

Listing Attributes

For now, list the attributes for the telephone-directory example with the entities with which you think they belong. Your list should look like Figure 2-11.

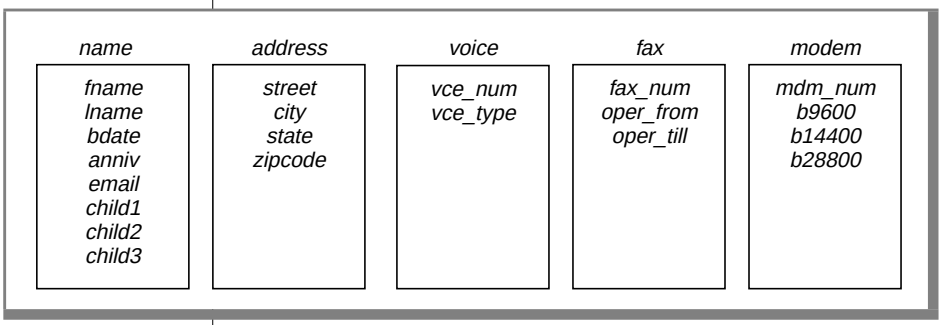


Figure 2-11 *Attributes for the Telephone-Directory Example*

About Entity Occurrences

An additional data object is the entity occurrence. Each row in a table represents a specific, single occurrence of the entity. For example, if *customer* is an entity, a **customer** table represents the idea of customer; in it, each row represents one specific customer, such as Sue Smith. Keep in mind that entities will become tables, attributes become columns, and entity occurrences become rows.

Diagramming Data Objects

Now you know and understand the entities and relationships in your database. That is the most important part of the relational-database design process. Once 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 uses the E-R diagram approach originally developed by C. R. Bachman. E-R diagrams serve the following purposes:

- Model the information needs 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 as well as 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 2-12 shows a sample E-R diagram.

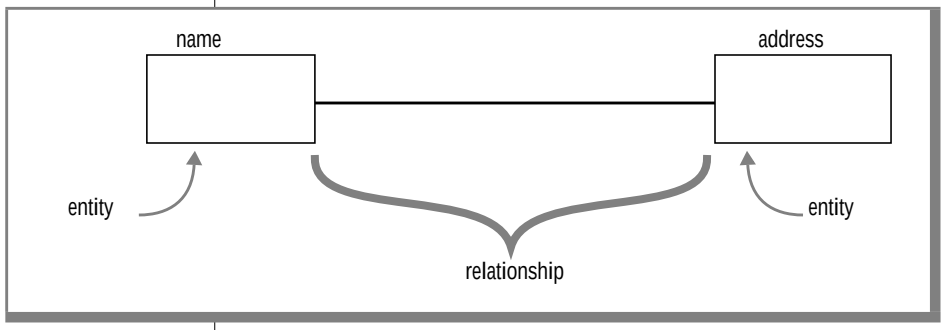


Figure 2-12
Symbols of an Entity-Relationship Diagram

In an E-R diagram, a box represents an entity. A line represents the relationships that connect the entities. In addition, Figure 2-13 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.

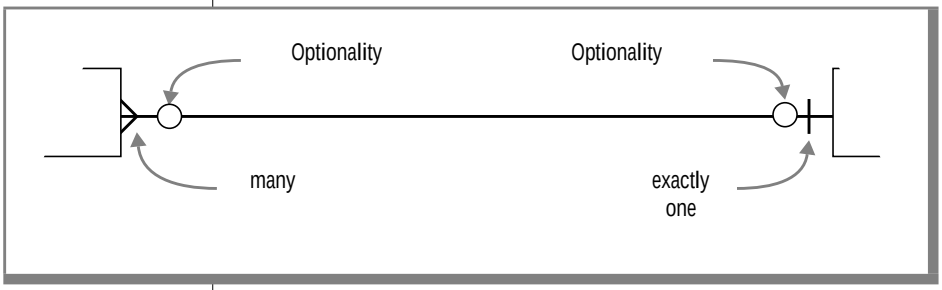


Figure 2-13
The Parts of a Relationship in an Entity-Relationship Diagram

Reading 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 Figure 2-14, 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.

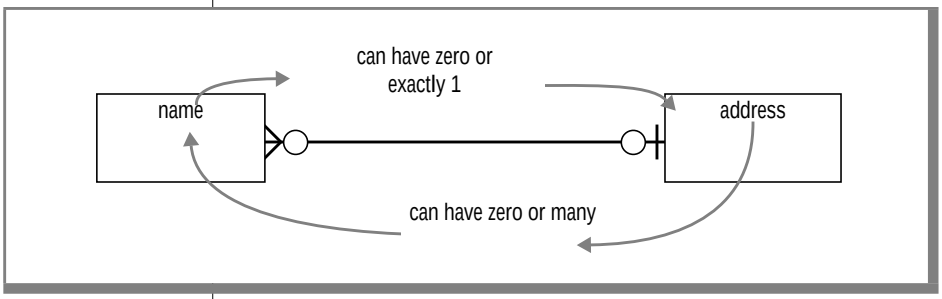


Figure 2-14
Reading an Entity-Relationship Diagram

Telephone-Directory Example

Figure 2-15 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 Figure 2-15 with the matrix in Figure 2-10 on page 2-16. Verify for yourself that the relationships are the same in both figures.

A matrix such as Figure 2-10 on page 2-16 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 Figure 2-15, and this type of diagram might be easier to read when you review an existing model.

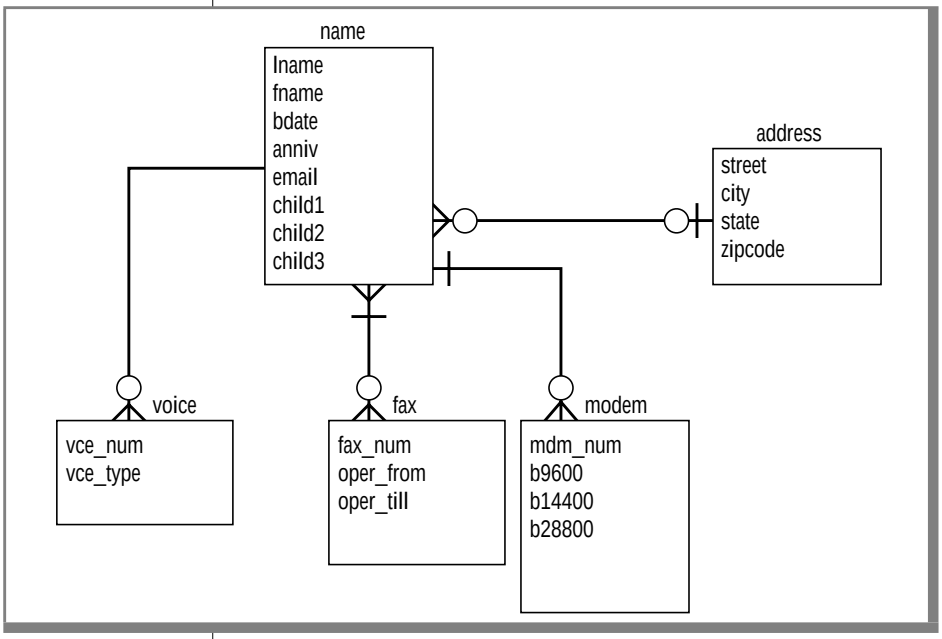


Figure 2-15 *Preliminary Entity-Relationship Diagram of the Telephone-Directory Example*

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

Chapter 4, "Implementing a Relational Data Model," shows you how to create a database from the E-R data model.

Translating 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 "Defining Tables, Rows, and Columns" on page 2-23.

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 discussed in "Normalizing a Data Model" on page 2-31.

After you normalize the data model, you can use SQL statements to create a database that is based on your data model. Chapter 4, "Implementing a Relational Data Model," describes how to create a database and provides the database schema for the telephone-directory example.

Each entity that you choose 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.

Defining 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.

Placing 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. The domain characteristics of a column can consist of the following items:

- 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, personnel number)
- Allowable values (for example, only grades S or U)
- Uniqueness
- Null support
- Default value
- Referential constraints

You define the domain characteristics when you create your tables. For information about how to define domains, see Chapter 3, "Choosing Data Types." For information about how to create your tables and database, see Chapter 4, "Implementing a Relational Data Model."

Determining 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.

The primary key should be a numeric data type (INT or SMALLINT), SERIAL data type, or a short character string (as used for codes). Informix recommends that you do not use long character strings as primary keys.

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.

Some entities have ready-made primary keys such as catalog codes or identity numbers, which are defined outside the model. These are user-assigned keys.

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. When you select the column of a candidate key, you know the result does not contain any duplicate rows, therefore, the result of a SELECT operation can be a table in its own right, with the selected candidate key as its primary key.

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 offers the SERIAL data type 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 could 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. Figure 2-16 shows the primary and foreign keys of the **customer** and **order** tables from the demonstration database.

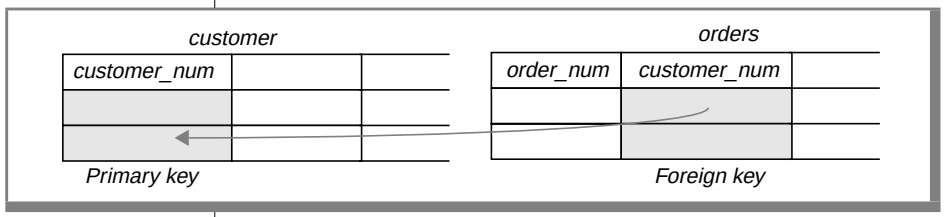


Figure 2-16
Primary and Foreign Keys in the Customer-Order Relationships

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 disallows deletes 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. Referential integrity is discussed in the *Informix Guide to SQL: Tutorial*.

Adding Keys to the Telephone-Directory Diagram

Figure 2-17 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 chosen. 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.

For the **voice**, **fax**, and **modem** tables, the telephone numbers are shown as primary keys. These tables are joined to the **name** table through the **rec_num** key.

The **address** table also uses a system-generated primary key, **id_num**. The **address** table must have a primary key because the business rules state that an address can exist when no names use it. If the business rules prevent an address from existing unless a name is associated with it, then the **address** table could be joined to the **name** table with the foreign key **rec_num** only.

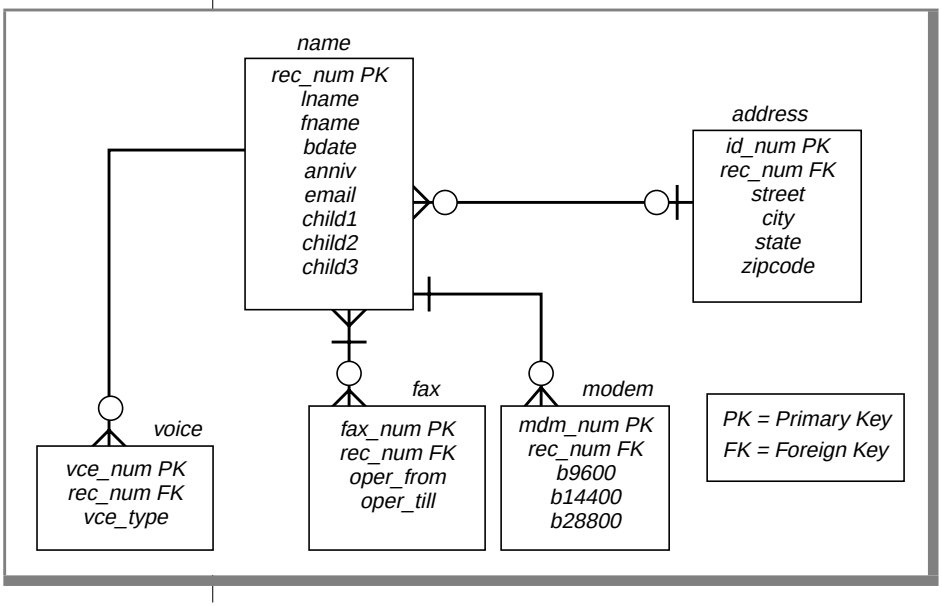


Figure 2-17
Telephone-Directory Diagram with Primary and Foreign Keys Added

Resolving 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 discussed in "Normalizing a Data Model" on page 2-31.

Resolving m:n Relationships

Many-to-many (m:n) relationships add complexity and confusion to your model and to the application development process. The key to resolve m:n relationships is to separate the two entities and create two one-to-many (1:n) relationships between them with a third *intersect* entity. The intersect entity usually contains attributes from both connecting entities.

To resolve a m:n relationship, analyze your business rules again. Have you accurately diagrammed the relationship? In the telephone-directory example, we have a m:n relationship between the *name* and *fax* entities as Figure 2-17 on page 2-28 shows. The business rules say: "One person can have zero, one, or many fax numbers; *a fax number can be for several people*." Based on what we selected earlier as our primary key for the *voice* entity, a m:n relationship exists.

A problem exists in the *fax* entity because the telephone number, which is designated as the primary key, can appear more than one time in the *fax* entity; this violates the qualification of a primary key. Remember, the primary key must be unique.

To resolve this m:n relationship, you can add an intersect entity between the *name* and *fax* entities, as Figure 2-18 shows. The new intersect entity, *faxname*, contains two attributes, **fax_num** and **rec_num**. The primary key for the entity is a composite of both attributes. Individually, each attribute is a foreign key that references the table from which it came. The relationship between the **name** and **faxname** tables is 1:n because one name can be associated with many fax numbers; in the other direction, each **faxname** combination can be associated with one **rec_num**. The relationship between the **fax** and **faxname** tables is 1:n because each number can be associated with many **faxname** combinations.

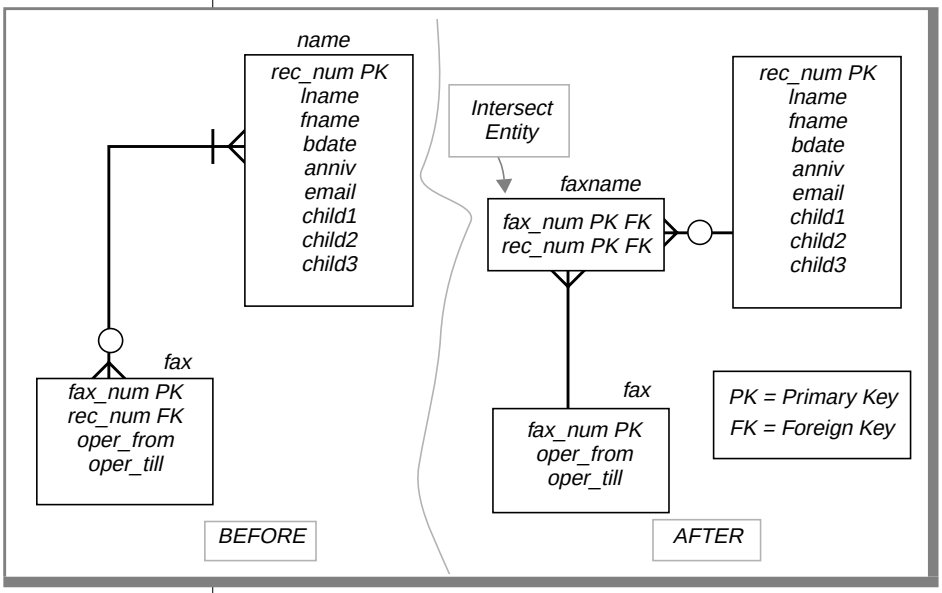


Figure 2-18
Resolving a Many-to-Many (m:n) Relationship

Resolving Other Special Relationships

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

- Complex relationships
- Recursive relationships
- Redundant relationships

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.

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 bill-of-materials (parts are composed of subparts) and organizational structures (employee manages other employees). For an extended example of a recursive relationship, see the *Informix Guide to SQL: Tutorial*. You might choose not to resolve recursive relationships.

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 need to add an entity to the model to resolve the redundancy. Your *Performance Guide* discusses additional topics that are related to redundancy in a data model.

Normalizing a Data Model

The telephone-directory example in this chapter appears to be a good model. You could 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:

- Produce greater flexibility in your design
- Ensure that attributes are placed in the proper 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 discusses the first three normal forms. Each normal form constrains the data to be more organized 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 telephone-directory example in Figure 2-19, it appears that the **name** table contains repeating columns, *child1*, *child2*, and *child3*.

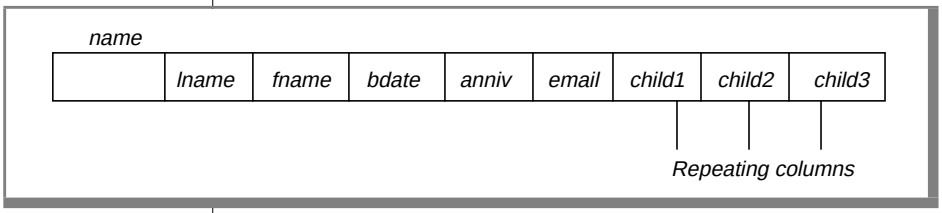


Figure 2-19
Name Entity Before Normalization

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 would have to 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 as Figure 2-20 shows. 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**.

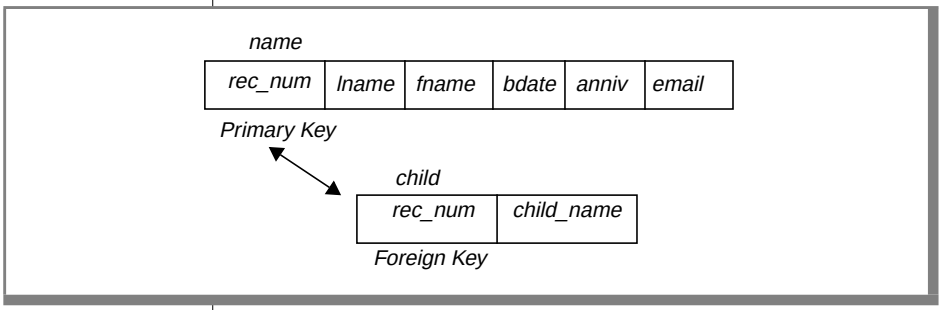


Figure 2-20
First Normal Form Reached for Name Entity

Now check Figure 2-17 on page 2-28 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**. Figure 2-21 shows the data model normalized through the first normal form.

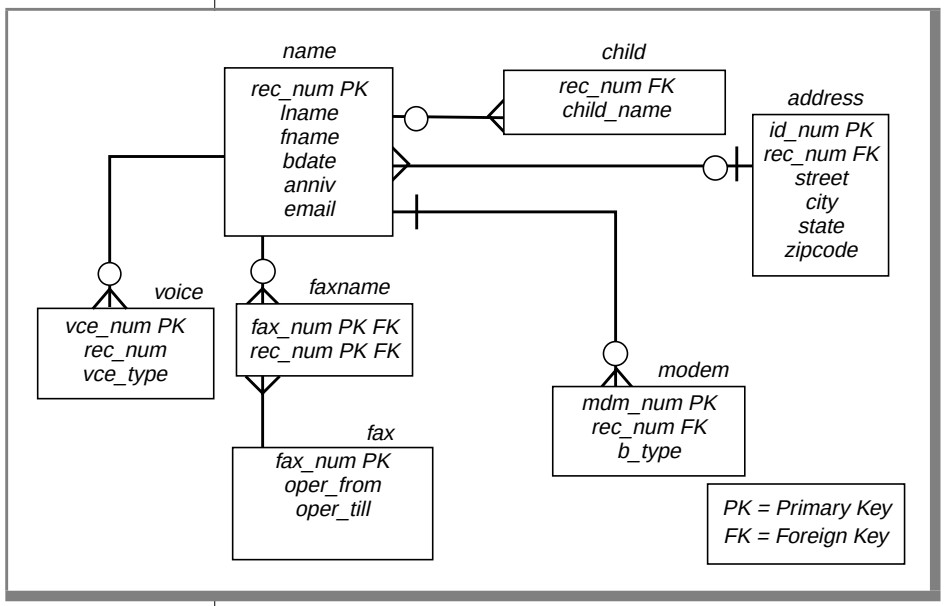


Figure 2-21
The Data Model of a Personal Telephone Directory

Second Normal Form

An entity is in the second normal form if it is in the first normal form and 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 discussed 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.

Chapter

3

Choosing Data Types

Defining the Domains 3-3
Data Types 3-4
Choosing a Data Type 3-4
Numeric Types 3-7
Chronological Types. 3-12
Character Types 3-17
Changing the Data Type 3-23
Null Values 3-23
Default Values 3-23
Check Constraints 3-24

Once a data model is prepared, 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 discusses the decisions that you must make to define the column data types and constraints.

The second step (see Chapter 4) is to use the CREATE DATABASE and CREATE TABLE statements to implement the model and populate the tables with data.

Defining the Domains

To complete the data model described in Chapter 2, "Building a Relational Data Model" 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, first define the *constraints* that a data value must satisfy before it can be part of the domain. You use the following constraints to specify a column domain:

- Data types
- Default values
- Check constraints

You can identify the primary and foreign keys in each table to place referential constraints on columns. Chapter 2, "Building a Relational Data Model" discusses how these keys are identified.

Data Types

The first constraint on any column is the one that is implicit in the data type for the column. When you choose a data type, you constrain the column so that it contains only values that can be represented by that 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 proper for that column but as few as possible of the values that are not proper for it.

Choosing a Data Type

Every column in a table must have a data type that is chosen from the types that the database server supports. The choice of data type is important for the following reasons:

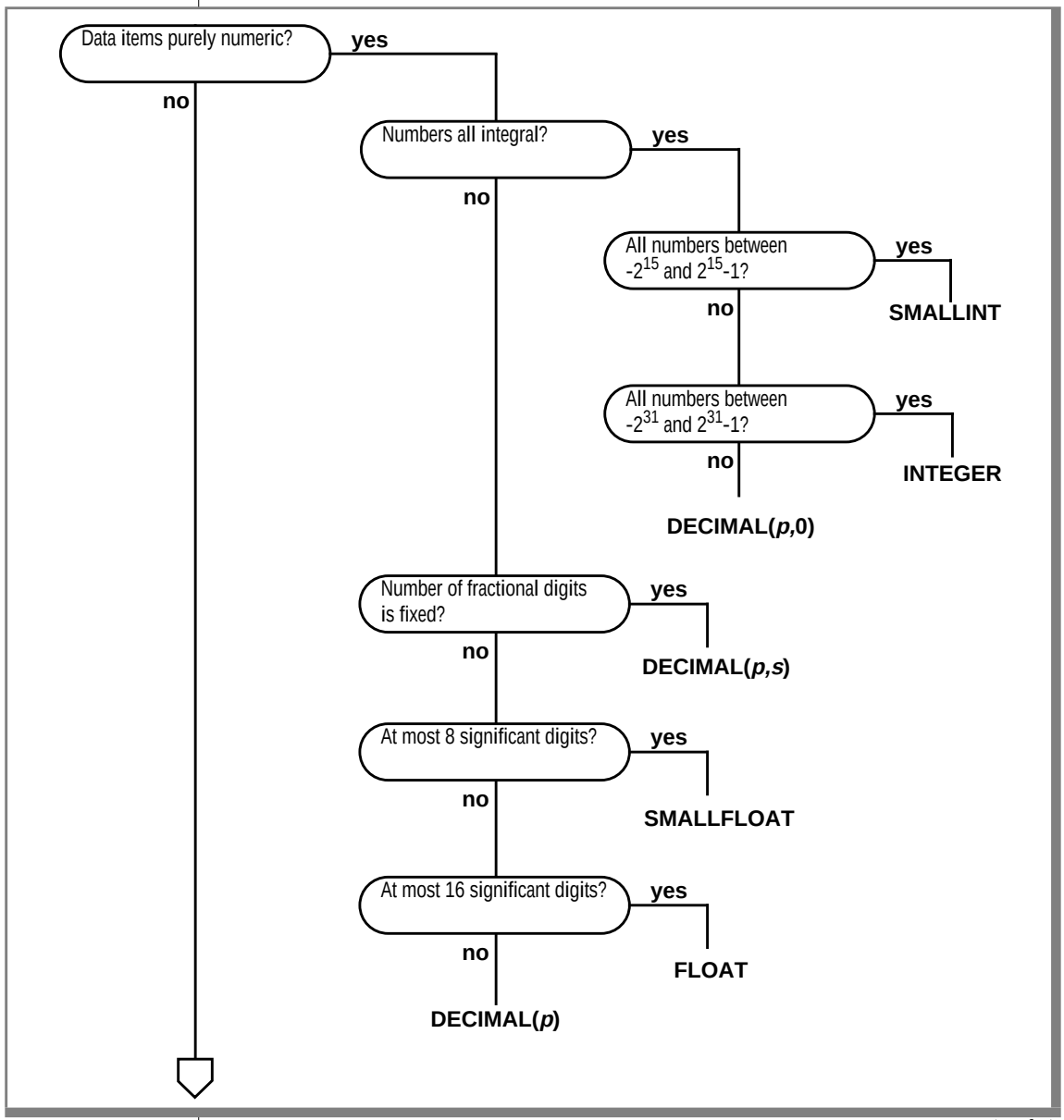
- It establishes the basic domain of the column; that is, the set of valid data items that the column can store.
- It determines the kinds of operations that you can perform on the data. For example, you cannot apply aggregate functions, such as SUM, to columns that are defined on a character data type.
- 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 tens or hundreds of thousands of rows. When a table reaches that many rows, the difference between a 4-byte and an 8-byte type can be crucial.

Using Data Types in Referential Constraints

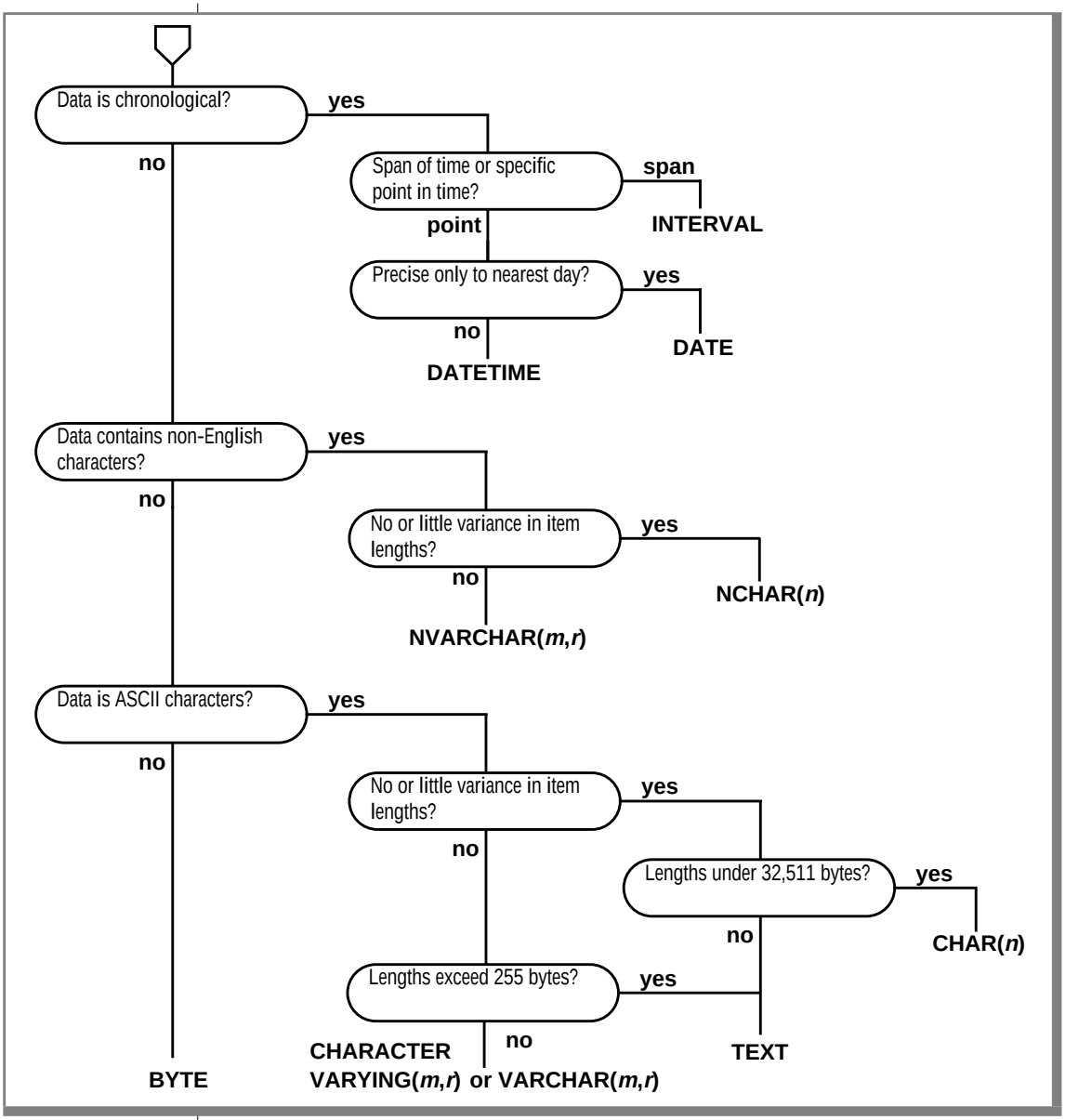
Almost all data type combinations must match when you are trying to pick columns for primary and foreign keys. For example, if you define a primary key as a CHAR data type, you must also define the foreign key as a CHAR data type. However, when you specify a SERIAL data type on a primary key in one table, you specify an INTEGER on the foreign key of the relationship. The SERIAL and INTEGER construction is the only data-type combination that you can mix in a relationship.

Figure 3-1 on page 3-5 shows the decision tree that summarizes the choices among data types. The choices are explained in the following sections.

Figure 3-1
Choosing a Data Type



(1 of 2)


Data is chronological?
yes
no
Span of time or specific point in time?
span
INTERVAL
point
Precise only to nearest day?
yes
DATE
no
DATETIME
Data contains non-English characters?
yes
no
No or little variance in item lengths?
yes
NCHAR(n)
no
NVARCHAR(m,r)
Data is ASCII characters?
yes
no
No or little variance in item lengths?
yes
no
Lengths under 32,511 bytes?
yes
CHAR(n)
no
Lengths exceed 255 bytes?
yes
no
TEXT
CHARACTER VARYING(m,r) or VARCHAR(m,r)
BYTE

(2 of 2)

Numeric Types

Informix database servers support eight numeric data types. Some are best suited for counters and codes, some for engineering quantities, and some for money.

Counters and Codes: 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 types are stored as signed binary integers. INTEGER values have 32 bits and can represent whole numbers from -2^{31} 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.

Automatic Sequences: SERIAL

The SERIAL data type is INTEGER with a special feature. Whenever you insert a new row into a table, the database server automatically generates a new value for a SERIAL column. A table can have only one SERIAL column. Because the database server generates them, the serial values in new rows are always different even when multiple users add rows at the same time. This service is useful, because it is difficult for an ordinary program to coin unique numeric codes under those conditions.

The database server uses all the positive serial numbers by the time it inserts 2^{31} rows in a table. You might not be concerned about exhausting the positive serial numbers because a single application would need to insert a row every second for 68 years, or 68 applications would need 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 continue to generate new numbers. It would treat the next serial quantity as a signed integer. Because the database server uses only positive values, it would wrap around and start to generate integer values that begin with a 1.

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 a SERIAL column are returned in the order in which they were created. That cannot be said of any other data type.

IDS

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

A SERIAL column is not automatically a unique column. To be sure that no duplicate serial numbers occur, you must apply a unique constraint (see "Using CREATE TABLE" on page 4-6). When you use DB-Access or Relational Object Manager to define the table, the database server automatically applies a unique constraint to any SERIAL column.

The SERIAL data type has the following advantages:

- It provides a convenient way to generate system-assigned keys.
- It produces unique numeric codes even when multiple users are update the table at the same time.
- Different tables can use different ranges of numbers.

The SERIAL data type has the following disadvantages:

- Only one SERIAL column is permitted in a table.
- It can produce only arbitrary numbers.

Altering the Next SERIAL Number

The starting value for a SERIAL column is set when the column is created (see "Using CREATE TABLE" on page 4-6). You can use the ALTER TABLE statement later to reset the *next* value, the value that is generated for the next-inserted row.

You cannot set the *next* value below the current maximum value in the column because doing so causes the database server to generate duplicate numbers in certain situations. However, you can set the *next* value to any value higher than the current maximum, which creates gaps in the sequence.

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 numerical quantity, fractional or whole, over a wide range of magnitudes from the cosmic to the microscopic. For example, they can easily represent both the average distance from the Earth to the Sun (1.5×10^{9} meters) or Planck's constant (6.625×10^{-27}). Their only restriction is their limited precision. Floating-point numbers retain only the most significant digits of their value. If a value has no more digits than a floating-point number can store, the value is stored exactly. If it has more digits, it is stored in approximate form, with its least-significant digits treated as zeros.

This lack of exactitude is fine for many uses, but never use a floating-point data type to record money or any other quantity whose least significant digits should not be changed to zero.

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. A FLOAT column retains about 16 digits of its values; a SMALLFLOAT column retains only about 8 digits.

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)

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 ranges from 10^{-129} to 10^{125}.

It is easy to be confused about decimal data types. The one under discussion is DECIMAL(*p*); that is, DECIMAL with only a precision specified. The size of DECIMAL(*p*) numbers depends on their precision; they occupy $1 + p/2$ bytes (rounded up to a whole number, if necessary).

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.

Fixed-Point Numbers: DECIMAL and MONEY

Most commercial applications need to store numbers that have fixed numbers of digits on the right and left of the decimal point. Amounts of money are the most common examples. Amounts in U.S. and other currencies are written with two digits to the right of the decimal point. Normally, you also know the number of digits needed on the left, depending on the kind 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. (Figure 3-2 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.

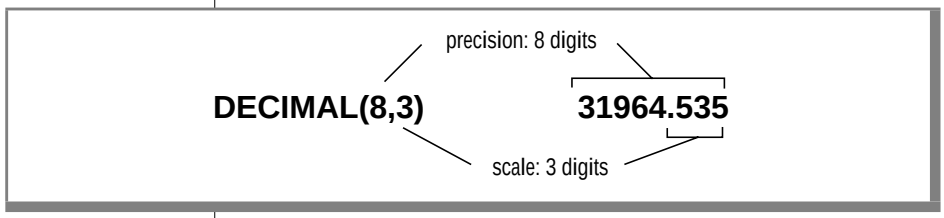


Figure 3-2 *The Relation Between Precision and Scale in a Fixed-Point Number*

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.

Choosing a currency format

GLS

Each nation has its own way to display money values. When an 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 locales, you can use the MONETARY category of the locale file to change the current format. For more information on how to use locales, see the *Informix Guide to GLS Functionality*. ♦

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

Chronological Types

Informix database servers support three data types to 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. Most often it holds a positive count of days into the current century.

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 prior to the epoch date; that is, a DATE value of `-1` represents the day December 30, 1899.

Because DATE values are integers, Informix database servers permit them to 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.

GLS

Choosing a date format

You can punctuate and order the components of a date in many ways. When an Informix database server 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:

```
10/25/95
```

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

For languages other than English, you can use the TIME category of the locale file to change the date format. For more information on how to use locales, refer to the *Informix Guide to GLS Functionality*.

Exact Points in Time: DATETIME

The DATETIME data type stores any moment in time in the era that begins 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*, and *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. Figure 3-3 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 "Forcing the format of a DATETIME or INTERVAL value" on page 3-16.)

Precision	Size*	Precision	Size*
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
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

* When *f* is odd, round the size to the next full byte.

Figure 3-3
Precisions for the DATETIME Data Type

Durations: 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).
- She has worked for 254 days (an INTERVAL value, the difference between the TODAY function and the starting DATE or DATETIME value).
- She begins work each day at 0900 hours (a DATETIME value).
- She works 8 hours (an INTERVAL value) with 45 minutes for lunch (another INTERVAL value).
- Her quitting time is 1745 hours (the sum of the DATETIME when she begins work and the two INTERVALs).

Like DATETIME, INTERVAL is a family of 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 Figure 3-4 shows.

Precision	Size*	Precision	Size*
year(*p*) to year	1 + *p* / 2	hour(*p*) to minute	2 + *p* / 2
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

* Round a fractional size to the next full byte.

Figure 3-4 *Precisions for the INTERVAL Data Type*

INTERVAL values can be negative as well as 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?"

Forcing 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 ...
```

GLS

Choosing a DATETIME Format

When an Informix database server 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:

```
1998-10-25 18:02:13
```

For languages other than English, you use the TIME category of the locale file to change the DATETIME format. For more information on how to use locales, see the *Informix Guide to GLS Functionality*.

To customize this DATETIME format, choose your locale appropriately or set the **GL_DATETIME** or **DBTIME** environment variable. For more information about these environment variables, see the *Informix Guide to GLS Functionality*.

GLS

Character Types

The database server supports several character 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 and non-English 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 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.

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 and non-English 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 less 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 single-byte ASCII space characters to make up *n* bytes.

Tip: *The database server accepts values that are extended with either single-byte or multibyte spaces as the locale defines.*

The database server sorts data in NCHAR(*n*) columns according to the order that the locale specifies. For example, the French locale specifies that the character *ê* is sorted after the value *e* but before the value *f*. In other words, the sort order dictated by the French locale is *e*, *ê*, *f*, and so on. For more information on how to use locales, refer to the *Informix Guide to GLS Functionality*.

Tip: *The only difference between CHAR(n) and NCHAR(n) data is how you sort and compare the data. You can store non-English 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 exists 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), and NVARCHAR(m,r)

For each of the following data types, *m* represents the maximum number of bytes and *r* represents the minimum number of bytes.

Tip: *The CHARACTER VARYING (m,r) data type is ANSI compliant. VARCHAR(M,R) is an Informix data type.*

Often the items in a character column are different lengths; that is, many are an average length, and only a few are the maximum length. The following 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*).** VARCHAR (*M,R*) is an Informix-specific data type for storing character data of varying length. In functionality, it is the same as CHARACTER VARYING(*M,R*).
- **NVARCHAR (*M,R*).** NVARCHAR (*M,R*) is also an Informix-specific data type for storing character data of varying length. It compares character data in the order that the locale specifies.

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 3-17*.

When you define columns of these data types, you 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 that 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 3-20.)

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

- It conserves disk space when the number of bytes that data items require vary 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 list describes the disadvantages of using varying-length data types:

- They do not allow lengths that exceed 255 bytes.
- Table updates can be slower in some circumstances.
- They are not available with all Informix database servers.

♦

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 a varying number 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.

AD/XP

Dynamic Server with AD and XP Options supports the TEXT data type in columns. However, Dynamic Server with AD and XP Options does not allow you to store a TEXT column in a blobspace or use a TEXT value in a stored procedure. ♦

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 *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. (See omu.)
- It is not available with all Informix database servers.

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.

AD/XP

Dynamic Server with AD and XP Options supports the BYTE data type in columns. However, Dynamic Server with AD and XP Options does not allow you to store a BYTE column in a blobspace or use a BYTE value in a stored procedure. ♦

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.

Using TEXT and BYTE Data Types

Collectively, columns of TEXT and BYTE data type are called *binary large objects*. The database server stores and retrieves them. 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 any SQL statement, interactive or programmed, a TEXT or BYTE column *cannot* 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, a TEXT or BYTE value can:

- be selected by name, optionally with a subscript to extract part of it.
- have its length returned by selecting LENGTH(*column*).
- be tested 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.

Changing 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

Columns in a table can be designated as containing 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 Chapter 2, "Building a Relational Data Model," 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 blank value.

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 need 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, you 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. 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*.

Chapter

4

Implementing a Relational Data Model

Creating the Database 4-3
Using CREATE DATABASE 4-4
Avoiding Name Conflicts 4-4
Selecting a Dbspace 4-5
Choosing the Type of Logging 4-5
Using CREATE TABLE 4-6
Using Synonyms with Table Names. 4-8
Using Synonym Chains 4-10
Using Command Scripts 4-11
Capturing the Schema 4-11
Executing the File 4-11
An Example 4-11
Populating the Tables. 4-12
Loading Source Data into a Table 4-12
Performing Bulk-Load Operations 4-13

This chapter shows how you use the SQL syntax to implement the data model that is described in Chapter 2, "Building a Relational Data Model." In other words, it shows you how to create a database and tables and populate the tables with data. This chapter also discusses database logging options, table synonyms, and command scripts:

Creating 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 discusses 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. However, the statements that create the model can be stored and executed automatically. For more information, see "Using Command Scripts" on page 4-11.

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.

Using CREATE DATABASE

A database is a container that holds all the 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.

GLS

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 on how to use alternative locales, see the *Informix Guide to GLS Functionality*. ♦

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

Avoiding 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. This is sometimes done, for example, to create a safe environment for testing apart from the operational data. In that case, be sure that you are using the correct database server when you create the database, and again when you access it later.)

Selecting 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. The administrator can put a database in a 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 *Administrator's Guide*. For information about how to fragment the tables of your database across multiple dbspaces, see Chapter 5, "Fragmentation Strategies" in this manual.

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

Choosing the Type of Logging

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

- **No logging at all.** Informix does not recommend this 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 choose 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.

  AD/XP

  Dynamic Server with AD and XP Options does not support nonlogging databases. However, Dynamic Server with AD and XP Options does support nonlogging tables. For more information about nonlogging tables, see "Logging and Nonlogging Tables for Dynamic Server with AD and XP Options" on page 7-13. ♦

- **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 crash. 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. See the discussion of ANSI-compliant databases in the *Getting Started* manual.

```
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.

The database server administrator can turn transaction logging on and off (except ANSI mode) later. For example, the administrator can turn it off before inserting a large number of new rows. For information about how to turn logging off for databases that are not ANSI-compliant, see your *Administrator's Guide*.

Using 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)
- If the column (or columns) is a primary key, the primary-key constraint

- If the column (or columns) is a foreign key, the foreign-key constraint
- If the column is not a primary key and should not allow nulls, the not null constraint
- If the column is not a primary key and should not allow duplicates, the unique constraint
- If the column has a default value, the default constraint
- If the column has a check constraint, the 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 Figure 2-21 on page 2-33. 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)
        );
```

Using Synonyms with Table Names

A *synonym* is a name that you can use in place of another name. You use the CREATE SYNONYM statement to provide an alternative name for a table or view.

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;
```

Once you create the synonym you can use it anywhere in the current database that you might use the original table name, 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.

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 need 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 stores7@nantes;
CREATE SYNONYM customer FOR stores7@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 translated through the synonym into a reference to the table that exists on the database server **avignon**.

Using 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 accesses to it? One way is to install a synonym to replace the old table, as the following example shows:

```
DATABASE stores7@avignon EXCLUSIVE;
RENAME TABLE customer TO old_cust;
CREATE SYNONYM customer FOR stores7@db_crunch:customer;
CLOSE DATABASE;
```

When you execute a query within **stores7@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 **stores7@nantes** still redirects references to **customer** to database **stores7@avignon**; however, the new synonym there sends the query to database **stores7@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). As long as the data model is the same, a program cannot tell the difference between a local database server and a remote one.

Using Command Scripts

You can enter SQL statements interactively to create the database and tables. In some cases, however, 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 commands to create a database in a file and execute them automatically.

Capturing the Schema

You can write the statements to implement your model into a file. However, you can also have a program that implements the model for you. 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 interactively, 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*.

Executing the File

Programs that you use to enter SQL statements interactively, such as DB-Access or Relational Object Manager, can be run from a file of commands. You can start DB-Access or Relational Object Manager to read and execute a file of commands that you or **dbschema** prepared. For more information, see the *DB-Access User Manual* or the documentation for Relational Object Manager.

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.

Populating the Tables

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

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

If you are preparing an application program in another language, you can use the program to enter rows.

Often, the initial rows of a large table can be derived from data that is stored in tables in another database or in operating-system files. You can move the data into your new database in a bulk operation. If the data is in another Informix database, you can retrieve it in several ways.

You can 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 could 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 stores7@otherserver:items
```

Loading Source Data into a Table

When the data source is another kind of file or 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 a file, you can use the **dbload** utility to load it into a table. For more information on **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*.

AD/XP

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

Performing 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.

  AD/XP

  You cannot turn off logging for Dynamic Server with AD and XP Options databases. However, you can create nonlogging tables (raw permanent or static permanent) in the database. ♦
- 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.

You can enclose the steps to populate a database in a script of operating-system commands. You can invoke the command-line equivalents to ON-Monitor to automate the database server administrator commands.

Chapter

5

Fragmentation Strategies

What Is Fragmentation? 5-3
Enhanced Fragmentation for Dynamic Server with AD and XP Options 5-6
Why Use Fragmentation? 5-6
Whose Responsibility Is Fragmentation? 5-8
Fragmentation and Logging 5-8

Distribution Schemes for Table Fragmentation 5-9
Round-Robin Distribution Scheme 5-11
Expression-Based Distribution Schemes 5-11
Range Rule 5-12
Arbitrary Rule 5-12
Using the MOD Function in an Expression-Based Distribution Scheme 5-13
Inserting and Updating Rows in Expression-Based Fragments 5-13
System-Defined Hash Distribution Scheme 5-13
Hybrid Distribution Scheme 5-14
When Can the Database Server Eliminate Fragments from a Search? 5-15
Query Expressions for Fragment Elimination 5-16
Expression-Based Distribution Schemes for Fragment Elimination 5-16
Summary of Fragment Elimination for Dynamic Server . . . 5-20
Summary of Fragment Elimination for Dynamic Server with AD and XP Options 5-21

Creating a Fragmented Table 5-26
Creating a New Fragmented Table 5-27
Creating a Fragmented Table from Nonfragmented Tables 5-28
Creating a Table from More Than One Nonfragmented Table 5-29
Creating a Fragmented Table from a Single Nonfragmented Table 5-29

Modifying a Fragmented Table 5-30
Modifying Fragmentation Strategies 5-30
Using the INIT Clause to Reinitialize a Fragmentation Scheme Completely 5-31
Using the MODIFY Clause to Modify an Existing Fragmentation Strategy for Dynamic Server 5-32
Using ATTACH and DETACH to Modify an Existing Fragmentation Strategy for Dynamic Server with AD and XP OPtions 5-33
Adding a New Fragment 5-34
Dropping a Fragment 5-35

Fragmenting Temporary Tables. 5-37
Fragmenting Temporary Tables for Dynamic Server with AD and XP Options 5-37
Defining Your Own Fragmentation Strategy 5-37
Letting Dynamic Server with AD and XP Options Define a Fragmentation Strategy 5-38

Fragmentation of Table Indexes 5-39
Attached Indexes 5-39
Detached Indexes 5-40
Rowids. 5-41
Creating a Rowid Column. 5-41
What Happens When You Create a Rowid Column?. 5-42

Accessing Data Stored in Fragmented Tables 5-42
Using Primary Keys Instead of Rowids. 5-42
Rowid in a Fragmented Table for Dynamic Server 5-43
Creating a Rowid Column in a Fragmented Table. 5-43
Granting and Revoking Privileges from Fragments 5-44

This chapter describes the different fragmentation strategies that your database server supports and shows how to implement fragmentation strategies. It discusses the following topics:

- What is fragmentation?
- Distribution schemes for table fragmentation
- Creating a fragmented table
- Modifying a fragmented table
- Fragmentation of temporary tables
- Fragmentation of table indexes
- Accessing data stored in fragmented tables

The information in this chapter applies to the following database servers only:

- Informix Dynamic Server
- Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options

For information about how to formulate a fragmentation strategy, see your *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 *partitions)* in a separate dbspace associated with a specific physical disk. You create the fragments and assign them to dbspaces with SQL statements.

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 discussed in your *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.

The database server supports the following distribution schemes:

- **Round-robin.** This type of fragmentation places rows one after another in fragments, rotating through the series of fragments to distribute the rows evenly.

  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.

- **Expression-based.** This type of fragmentation 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.

AD/XP

- **System-defined hash.** This type of fragmentation uses an internal, system-defined rule that distributes rows with the objective of keeping the same number of rows in each fragment.

- **Hybrid.** This type of fragmentation combines two distribution schemes. The *primary* distribution scheme chooses the dbslice. The *secondary* distribution scheme puts rows in specific dbspaces within the dbslice.The dbspaces usually reside on different coservers. ♦

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.

The database server stores the location of each table and index fragment, along 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. For the complete listing of the information that this system catalog table contains, see the *Informix Guide to SQL: Reference*.

Because the database server has information on which fragments contain which data, the database server can route client requests for data to the appropriate fragment without accessing irrelevant fragments, as Figure 5-1 illustrates. (This statement is not true for the round-robin distribution scheme or all expression-distribution schemes, however. For more information, see "Distribution Schemes for Table Fragmentation" on page 5-9.)

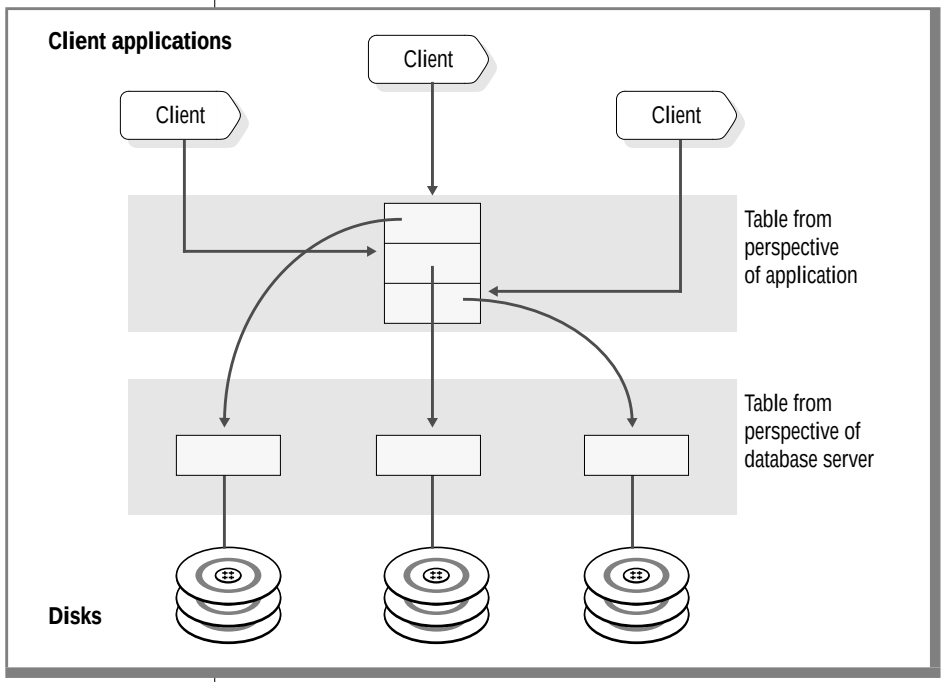


Figure 5-1
Database Server Routes Requests for Data from Client Applications

AD/XP

Enhanced Fragmentation for Dynamic Server with AD and XP Options

Dynamic Server with AD and XP Options can fragment tables across disks that belong to different coservers. Each table fragment can reside in a separate dbspace that is associated with physical disks that belong to different coservers. A *dbslice* provides the mechanism to manage many dbspaces across multiple coservers. Once you create the dbslices and dbspaces, you can create tables that are fragmented across multiple coservers.

For information on the advantages of fragmenting tables across coservers, see your *Performance Guide*. For information about how to create dbslices and dbspaces, see your *Administrator's Guide*.

Why Use Fragmentation?

Consider fragmenting your tables if you have at least one of the following goals:

- Improve single-user response time

  You can improve the performance of individual queries if you use fragmentation with parallel database query (PDQ) to scan fragments spread across multiple disks in parallel (Dynamic Server and Dynamic Server with AD and XP Options) or across multiple coservers in parallel (Dynamic Server with AD and XP Options). For a description of PDQ, see your *Administrator's Guide*. For performance considerations with PDQ and fragmentation, see your *Performance Guide*.

- Improve concurrency

  Fragmentation can reduce contention for data located in large, high-use tables. Fragmentation reduces contention because each fragment resides on a separate I/O device, and the database server directs queries to the appropriate fragment.

- Improve availability

  You can improve the availability of your data if you use fragmentation. If a fragment becomes unavailable, the database server is still able to access the remaining fragments. For information about how to skip inaccessible fragments, see your *Administrator's Guide*.

- Improve backup-and-restore characteristics

  Fragmentation gives you a finer backup-and-restore granularity. This granularity can reduce the time it takes for backup-and-restore operations. In addition, if you use ON-Bar or ON-Archive, you can perform backup-and-restore operations in parallel to improve performance. For more information about the ON-Bar backup and restore system, see your *Backup and Restore Guide*.

  IDS

  For more information on ON-Archive, refer to the *Archive and Backup Guide*. ♦

  AD/XP

- Improve data-load performance

  To improve the performance of loading very large databases, use fragmentation with *external tables*. For more information on how to use *external tables* to load databases, see your *Administrator's Guide*.

  When Dynamic Server with AD and XP Options uses *external tables* to load a table that is fragmented across multiple coservers, it allocates threads to load the data into the fragments in parallel. You can expect a near-linear improvement in performance with each fragment (located on a separate coserver) that you add to a table. ♦

Each of the preceding goals has its own implications for the fragmentation strategy that you ultimately implement. For more information about these goals, see the discussion about how to formulate a fragmentation strategy in your *Performance Guide*.

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.

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. This schema can include table fragmentation. The database server administrator, on the other hand, is responsible for allocating the disk space in which the fragmented tables will reside. Because neither of these responsibilities can be performed in isolation of the other, to implement fragmentation requires a cooperative effort between the database server administrator and the DBA. 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 *Administrator's Guide* and *Performance Guide*.

Fragmentation and Logging

IDS

With Dynamic Server, a fragmented table 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. ♦

AD/XP

With Dynamic Server with AD and XP Options, fragmented tables always belong to a database that uses logging. However, Dynamic Server with AD and XP Options does support several logging and nonlogging table types. For more information, see "Logging and Nonlogging Tables for Dynamic Server with AD and XP Options" on page 7-13. ♦

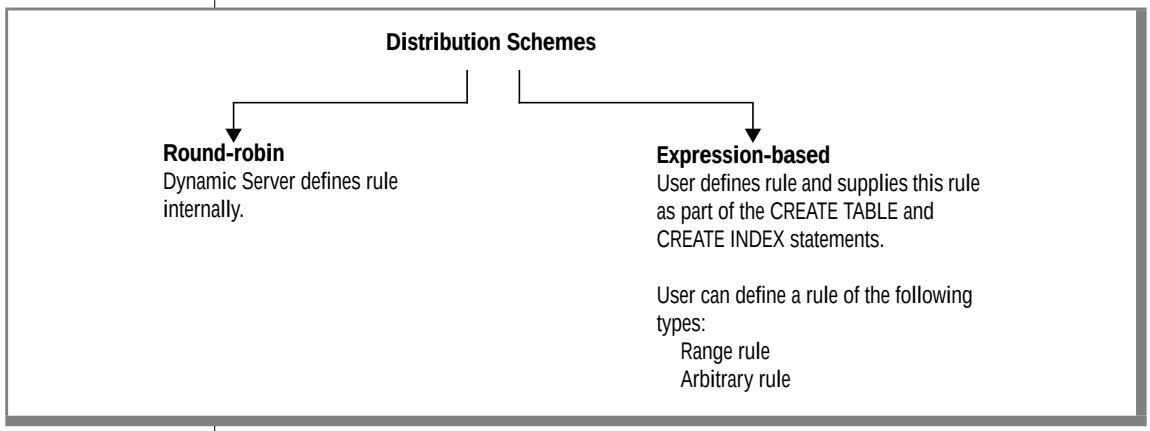
Distribution Schemes for Table Fragmentation

A distribution scheme is a method that the database server uses to distribute rows or index entries to fragments. Dynamic Server and Dynamic Server with AD and XP Options support different distribution schemes.

IDS

Figure 5-2 shows the distribution schemes that Dynamic Server supports.

Figure 5-2
Distribution Schemes and Fragmentation Rules for Dynamic Server

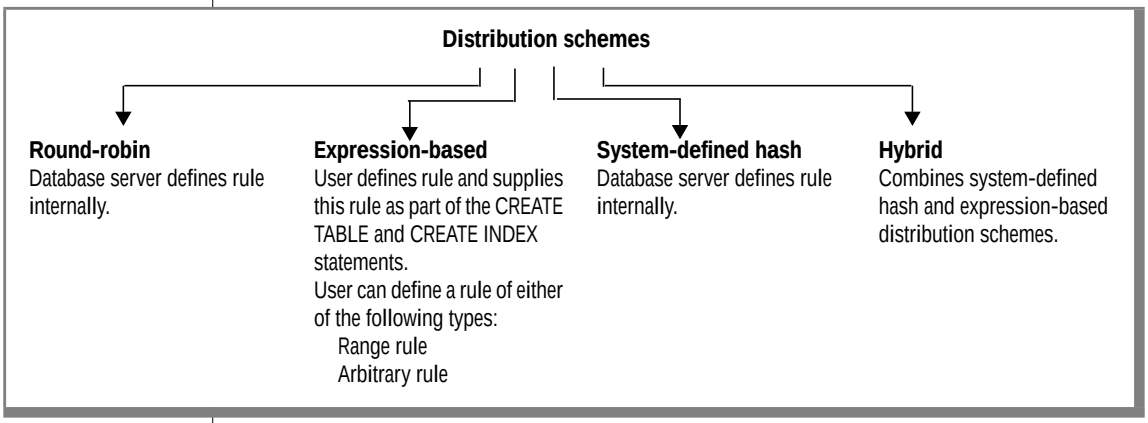


-

AD/XP

Figure 5-3 shows the distribution schemes that Dynamic Server with AD and XP Options supports.

Figure 5-3
Distribution Schemes and Fragmentation Rules for Dynamic Server with AD and XP Options



♦

The following sections describe the distribution schemes available for your database server.

For complete descriptions of the SQL syntax for the distribution schemes, see the CREATE TABLE and CREATE INDEX statements in the *Informix Guide to SQL: Syntax*.

For information about how to formulate fragmentation strategies, see your *Performance Guide*.

Round-Robin Distribution Scheme

To specify a round-robin distribution scheme, use the FRAGMENT BY ROUND ROBIN clause of the CREATE TABLE statement, as follows:

```
CREATE TABLE account...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. Round-robin distributions are also called *even distributions* because information is distributed evenly among the fragments. The rule for distributing rows to tables that use round-robin distribution is internal to the database server.

Important: *You can only use the round-robin distribution scheme to fragment table rows. You cannot fragment a table index with this distribution scheme. See* *"Fragmentation of Table Indexes" on page 5-39**.*

Expression-Based Distribution Schemes

To use an expression-based distribution scheme, use the FRAGMENT BY EXPRESSION clause of the CREATE TABLE or CREATE INDEX statement. This clause takes the following form:

```
CREATE TABLE tablename ... FRAGMENT BY EXPRESSION
    <expression_1> IN dbspace_1
    <expression_2> IN dbspace_2
    .
    .
    .
    <expression_n> IN dbspace_n;
```

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 all three 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 operator AND

A range rule can refer to only one column in a table but can make multiple references to this column. You define one expression for each of the fragments in your table, as shown in the following example:

```
.
.
.
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
```

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 shown in the following example:

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

IDS

Using the MOD Function in an Expression-Based Distribution Scheme

With Dynamic Server, 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
```

Inserting and Updating Rows in Expression-Based Fragments

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 specified by the remainder clause. 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.

AD/XP

System-Defined Hash Distribution Scheme

In a system-defined hash distribution scheme, Dynamic Server with AD and XP Options balances the load between the specified dbspaces as you insert records and distributes the rows in such a way that the fragments maintain approximately the same number of rows.

To specify a system-defined hash distribution scheme, use the FRAGMENT BY HASH clause in the CREATE TABLE statement as follows:

```
CREATE TABLE tablename...FRAGMENT BY HASH (column list)
    IN dbspace1, dbspace2, dbspace3
```

In a system-defined hash distribution scheme, specify at least two dbspaces where you want the fragments to be placed or specify a dbslice.

You can also specify a composite key for the system-defined hash distribution scheme.

When Dynamic Server with AD and XP Options receives a request to insert a number of rows into a table that is fragmented by system-defined hash, it distributes the rows in such a way that the number of rows in each of the fragments remains approximately the same. The rule for distributing rows to tables fragmented by system-defined hash is internal to Dynamic Server with AD and XP Options.

Tip: *For very large databases that are fragmented across many coservers, Informix recommends that you fragment by system-defined hash on the key that is used to join two tables. For information about fragmenting tables across coservers and co-located joins, see your Performance Guide.*

For descriptions of the SQL syntax for the system-defined hash distribution scheme, refer to the ALTER FRAGMENT and CREATE TABLE statements in the *Informix Guide to SQL: Syntax*.

AD/XP

Hybrid Distribution Scheme

A *hybrid* distribution scheme is a combination of system-defined hash and expression-based distribution schemes. To specify a hybrid distribution scheme, use the FRAGMENT BY HYBRID clause of the CREATE TABLE statement. This clause takes the following form:

```
... FRAGMENT BY HYBRID (column list)
    EXPRESSION
        <expression_1> IN dbslice_1,
        <expression_2> IN dbslice_2,
        :
        :
        <expression_n> IN dbslice_n
```

When the database server receives a request to insert a number of rows into a table that uses hybrid fragmentation, it distributes the rows as follows:

- It uses the expression-based portion of the distribution scheme to determine the dbslice in which to store the row.
- It uses the hash portion of the hybrid distribution scheme to determine the dbspace (of the predetermined dbslice) in which to store the row.

The following CREATE TABLE statement includes a hybrid distribution scheme based on two columns of the table. When you insert a value into the table, the database server uses the value of **col_2** to determine the dbslice that stores the row. The database server generates a hash value for **col_1** that determines the dbspace within the dbslice that stores the row.

```
CREATE TABLE my_table (col_1 INT, col_2 DATE, col_3 CHAR(4))
    HYBRID (col_1) EXPRESSION
    col_2 < '01/01/1996' IN dbsl_1,
    col_2 >= '01/01/1996' AND col_2 < '01/01/1997' IN dbsl_2,
    REMAINDER IN dbsl_3;
```

When Can the Database Server Eliminate Fragments from a Search?

If you use an appropriate distribution scheme, the database server can eliminate fragments from a search before it performs the actual search. This capability can improve performance significantly and reduce contention for the disks on which fragments reside.

Whether the database server can eliminate fragments from a search depends on two factors:

- The form of the query expression (the expression in the WHERE clause)
- The distribution scheme of the table that is being searched

Query Expressions for Fragment Elimination

You can use two types of query expressions to eliminate fragments:

- Range expressions
- Equality expressions

Consider the following query expression template:

```
column operator value
```

Range expressions use a relational operator (<, >, <=, >=) and equality expressions use an equality operator (=, IN). The *value* in either type of query expression must be a literal, a host variable, a stored procedure variable, or a noncorrelated subquery.

AD/XP

With Dynamic Server with AD and XP Options, the *value* in a range or equality expression must be a literal or a host variable. The value cannot be a stored procedure variable or a noncorrelated subquery. ♦

Expression-Based Distribution Schemes for Fragment Elimination

The database server cannot eliminate fragments when you fragment a table with a round-robin distribution scheme. Furthermore, depending on the database server you use, not all expression-based distribution schemes give you the same fragment-elimination behavior. The following sections discuss the relationship between specific fragmentation rules and fragment-elimination behavior.

Nonoverlapping Fragments on a Single Column

A fragmentation rule that creates nonoverlapping fragments on a single column can eliminate fragments for queries with range expressions as well as queries with equality expressions. Figure 5-4 gives an example of a fragmentation rule that creates nonoverlapping fragments on a single column.

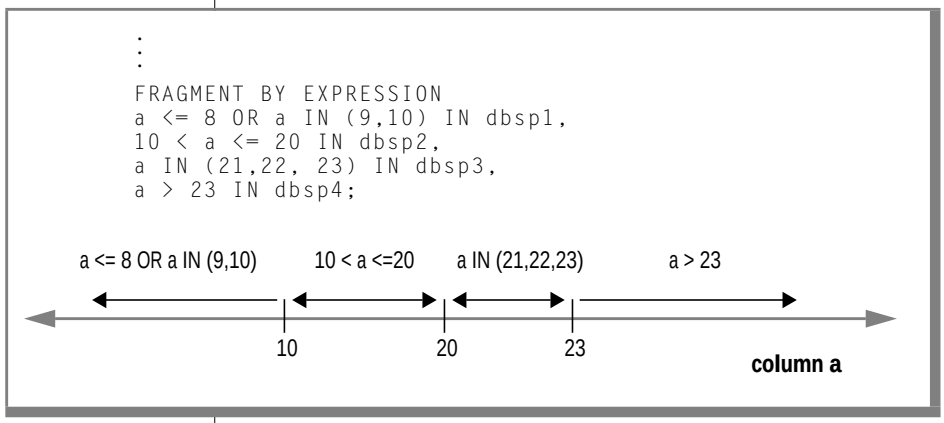


Figure 5-4
Schematic Example of Nonoverlapping Fragments on a Single Column

To create nonoverlapping fragments, use a range rule or an arbitrary rule that is based on a single column. You can use relational operators, as well as AND, IN, OR, and BETWEEN. However, use the BETWEEN operator with caution. When the database server parses the BETWEEN keyword, it includes the end points that you specify in the range of values.

IDS

With Dynamic Server, a fragmentation rule that creates nonoverlapping fragments on a single column is the preferred fragmentation rule from a fragment-elimination standpoint because Dynamic Server can eliminate fragments from queries with range expressions and equality expressions. Do not use a REMAINDER clause in the expression because the database server is not always able to eliminate the remainder fragment. ♦

Overlapping Fragments on a Single Column

The only restriction for this category of fragmentation rule is that you base the fragmentation rule on a single column. The fragments can be overlapping and noncontiguous. You can use any range or arbitrary rule that is based on a single column. Figure 5-5 gives an example of this type of fragmentation rule.

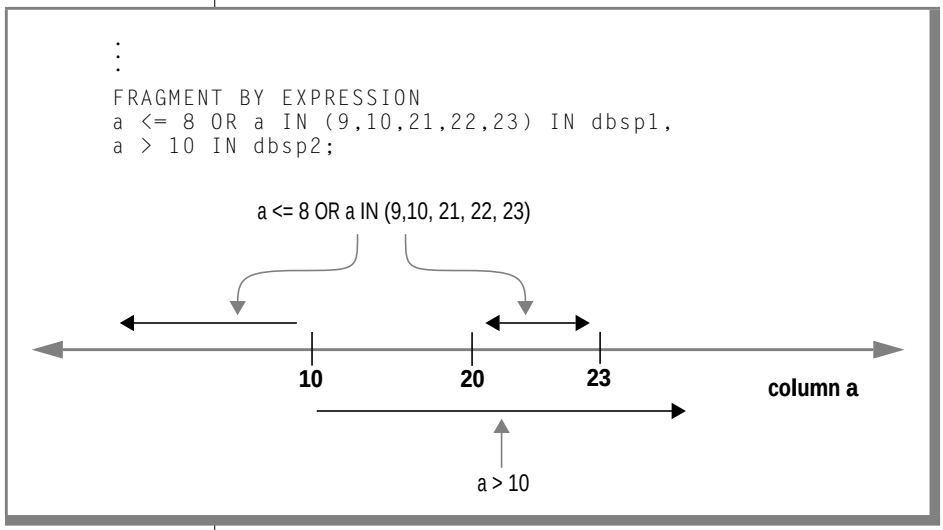


Figure 5-5
Schematic Example of Overlapping Fragments on a Single Column

If you use this type of distribution scheme, the database server loads rows into the first dbspace for which the expression fits.

IDS

For this category of fragmentation rule, Dynamic Server can eliminate fragments on an equality search but not a range search. However, the capability to eliminate fragments on an equality search can be useful because all INSERT and many UPDATE operations perform equality searches.

Overlapping fragments on a single column is acceptable if you cannot use an expression that creates nonoverlapping fragments with contiguous values. For example, in cases where a table is growing over time, you might want to use the MOD function in an expression-based distribution scheme to keep the fragments of similar size. Expression-based distribution schemes that use the MOD function in this way fall into this category because the values in each fragment are not contiguous. ♦

AD/XP

Dynamic Server with AD and XP Options can eliminate fragments on an equality search or range search. ♦

Nonoverlapping Fragments, Multiple Columns

This category of fragmentation rule uses an arbitrary rule to define nonoverlapping fragments based on multiple columns. This alternative is acceptable if you cannot obtain sufficient granularity with an expression based on a single column. Figure 5-6 gives an example of nonoverlapping fragments on two columns.

```
:
:
FRAGMENT BY EXPRESSION
0 < a <= 10 AND b IN ('E', 'F','G') IN dbsp1,
0 < a <= 10 AND b IN ('H', 'I','J') IN dbsp2,
10 < a <= 20 AND b IN ('E', 'F','G') IN dbsp3,
10 < a <= 20 AND b IN ('H', 'I','J') IN dbsp4,
20 < a <= 30 AND b IN ('E', 'F','G') IN dbsp5,
20 < a <= 30 AND b IN ('H', 'I','J') IN dbsp6;
```

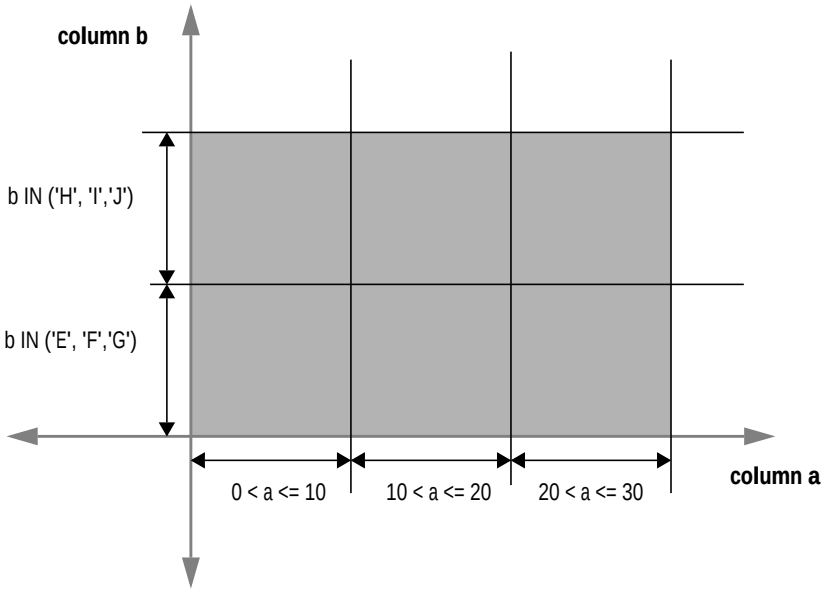


Figure 5-6
Schematic Example of Nonoverlapping Fragments on Two Columns

IDS

With this type of distribution scheme, Dynamic Server can eliminate fragments on an equality search but not a range search. Again, the capability to eliminate fragments on equality searches can be useful because all INSERT operations and many UPDATE operations perform equality searches. Do not use a REMAINDER clause in the expression because the database server is not always able to eliminate the remainder fragment. ♦

AD/XP

With this type of distribution scheme, Dynamic Server with AD and XP Options can eliminate fragments on equality searches and range searches. Dynamic Server with AD and XP Options can also eliminate the remainder fragment. ♦

IDS

Summary of Fragment Elimination for Dynamic Server

Figure 5-7 summarizes Dynamic Server fragment-elimination behavior for different combinations of distribution schemes and query expressions.

Figure 5-7
Fragment-Elimination Behavior for Dynamic Server

Distribution Scheme	Range Expression in Query	Equality Expression in Query
Nonoverlapping fragments on a single column	Can eliminate fragments	Can eliminate fragments
Overlapping or noncontiguous fragments on a single column	Cannot eliminate fragments	Can eliminate fragments
Nonoverlapping fragments on multiple columns	Cannot eliminate fragments	Can eliminate fragments

Figure 5-7 indicates that Dynamic Server eliminates fragments, but the WHERE clause of the query determines which fragments, if any, can be eliminated. For example, consider a table that is fragmented with the following expression:

```
.
.
.
FRAGMENT BY EXPRESSION
column_a > 100 AND column_b < 0 IN dbsp1,
column_a <= 100 AND column_b < 0 IN dbsp2,
column_b >= 0 IN dbsp3
```

In a WHERE clause that has the following expression, Dynamic Server cannot eliminate any of the fragments from the search:

```
column_a = 5 OR column_b = -50
```

In a WHERE clause that has the following expression, Dynamic Server eliminates the last fragment:

```
column_b = -50
```

In a WHERE clause that has the following expression, Dynamic Server eliminates the first and last fragments

```
column_a = 5 AND column_b = -50
```

For more information, see the FRAGMENT BY clause of the CREATE TABLE statement in the *Informix Guide to SQL: Syntax*.

AD/XP

Summary of Fragment Elimination for Dynamic Server with AD and XP Options

Dynamic Server with AD and XP Options can handle one- or two-column fragment elimination on queries with any combination of the following operators in the WHERE clause:

```
<, <=, >, >=, !=, IN
AND, OR, NOT
IS NULL, IS NOT NULL
MATCH, LIKE (where pattern has a fixed prefix)
```

Dynamic Server with AD and XP Options supports fragment elimination on all column types except columns that are defined on the BYTE and TEXT data types. Figure 5-8 on page 5-22 summarizes Dynamic Server with AD and XP Options fragment-elimination behavior for different combinations of distribution schemes and query expressions.

Figure 5-8
Fragment-Elimination Behavior for Dynamic Server with AD and XP Options

Distribution Scheme	Range Expression in Query	Equality Expression in Query
Nonoverlapping fragments on one or two columns	Can eliminate fragments	Can eliminate fragments
Overlapping or noncontiguous fragments on one or two columns	Can eliminate fragments	Can eliminate fragments
Hybrid (hash and expression-based)	Can eliminate fragments	Can eliminate fragments
REMAINDER clause in fragment	Can eliminate fragments	Can eliminate fragments
System-defined hash	Cannot eliminate fragments	Can eliminate fragments

Expression-Based Distribution Scheme and Query Expressions

Figure 5-8 shows that Dynamic Server with AD and XP Options can eliminate fragments for most combinations of distribution scheme and range or equality expression, but the WHERE clause of the query in question determines which fragments can be eliminated. For example, consider a table that is fragmented with the following expression:

```
.
.
.
FRAGMENT BY EXPRESSION
column_a > 100 AND column_b < 0 IN dbspace_1,
column_a <= 100 AND column_b < 0 IN dbspace_2,
column_b >= 0 IN dbspace_3,
REMAINDER IN dbspace_4
```

In a WHERE clause that has the following expression, Dynamic Server with AD and XP Options eliminates the third and fourth fragments:

```
column_b <= -10
```

In a WHERE clause that has the following expression, Dynamic Server with AD and XP Options eliminates the first, second, and fourth fragments:

```
column_a <= 100 AND column_b = -50
```

In a WHERE clause that has the following expression, Dynamic Server with AD and XP Options eliminates the first and second fragments:

```
column_a IS NULL
```

For more information, see the FRAGMENT BY clause of the CREATE TABLE statement in the *Informix Guide to SQL: Syntax*.

System-Defined Hash Distribution Scheme and Query Expressions

With a system-defined hash distribution scheme, Dynamic Server with AD and XP Options can eliminate fragments when the WHERE clause includes an equality expression.

Suppose you create the following table and a system-defined hash distribution scheme that fragments a table across multiple dbspaces within a dbslice:

```
CREATE TABLE account
  (account_num integer,
   account_bal integer,
   account_date date,
   account_name char(30)
  ) FRAGMENT BY HASH(account_num)
    IN acct_dbslc;
```

The database server cannot eliminate any of the fragments from the search if your WHERE clause includes a range expression, such as the following expression:

```
account_num >= 11111 AND 12345 <= account_num
```

However, the database server *can* eliminate any of the fragments from the search if your WHERE clause includes an equality expression, such as the following expression:

```
account_num = 12345
```

In this case, all but one of the dbspaces within the dbslice are eliminated from the search.

Hybrid Distribution Scheme and Query Expressions

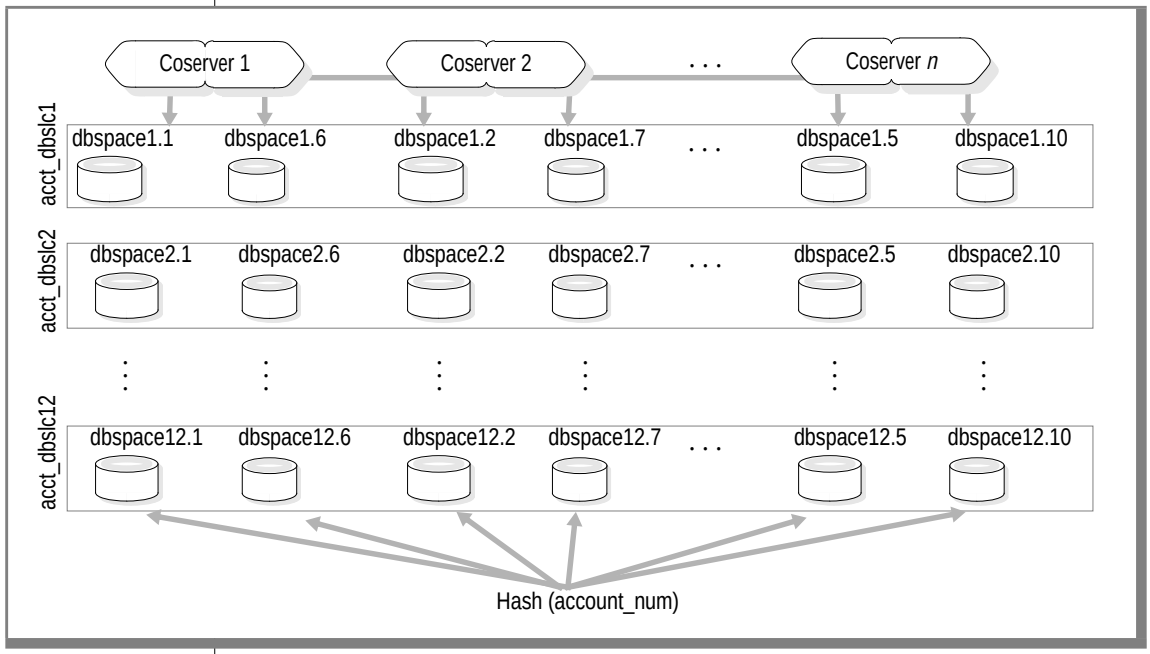
With a hybrid distribution scheme, Dynamic Server with AD and XP Options can eliminate fragments based on the system-defined hash distribution scheme, the expression-based distribution scheme, or both schemes.

Suppose you create the **account** table from the preceding example with the following hybrid distribution scheme:

```
FRAGMENT BY HYBRID (HASH (account_num)) EXPRESSION
    account_date < '01/01/1996' IN acct_dbslc1
    account_date < '02/01/1996' IN acct_dbslc2
    :
    account_date < '12/01/1996' IN acct_dbslc12
```

Figure 5-9 shows how you might layout the dbspaces within each of the 12 dbslices.

Figure 5-9
Hybrid Fragmentation and Fragment Elimination in Dynamic Server with AD and XP Options



When you define a hybrid distribution scheme, the database server can eliminate fragments from the search based on the system-defined hash distribution scheme, the expression-based distribution scheme, or both schemes.

The database server cannot eliminate fragments that the hash distribution scheme defines if your WHERE clause includes a range expression with this hash column, such as the following expression:

```
account_num >= 11111 AND 12345 <= account_num
```

However, the database server *can* eliminate any of the fragments from the search if your WHERE clause includes any combination of the following expressions:

- an equality expression on the hash column, such as the following expression:

  ```
  account_num = 12345
  ```

- an equality expression on the column used for the expression-based distribution scheme:

  ```
  account_date = '01/01/1996'
  ```

- a range expression with the column used for the expression-based distribution scheme:

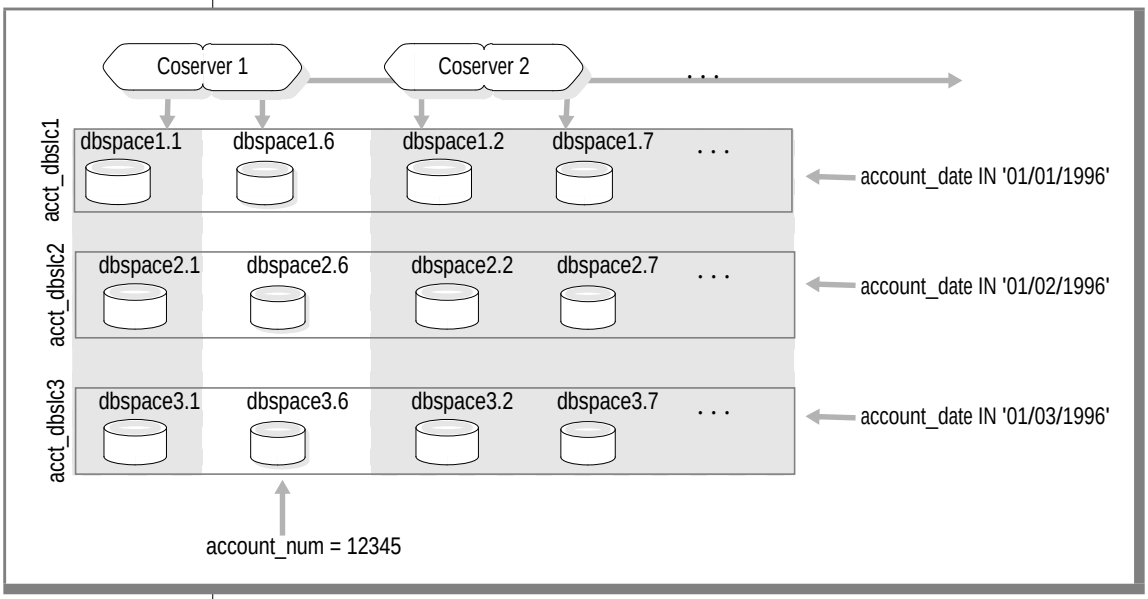
  ```
  account_date >= '01/01/1996'
      AND '03/01/1996' <= account_date
  ```

Suppose a WHERE clause in a query includes the following expression:

```
account_date IN ('01/01/1996','01/02/1996','01/03/1996')
    AND account_num = 12345
```

Dynamic Server with AD and XP Options eliminates the fourth through twelfth dbslices and all but three dbspaces (one dbspace from each of the three remaining dbslices) from the search for this query expression. The shaded areas of Figure 5-10 on page 5-26 show the fragments that the database server eliminates from the search.

Figure 5-10
Example of Fragment Elimination with Hybrid Fragmentation



Creating 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 database server performance guide.

Creating a New Fragmented Table

To create a fragmented table, use the FRAGMENT BY clause of the CREATE TABLE statement. Suppose that you wish to create a fragmented table similar to the **orders** table of the demonstration database. You decide on a round-robin distribution scheme with three fragments. Consult with your database server administrator to set up three dbspaces, one for each of the fragments: **dbspace1**, **dbspace2**, and **dbspace3**. To create the fragmented table, execute the following SQL statement:

```
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
```

IDS

If the table resides in a Dynamic Server database, you might decide instead to create the table with an expression-based distribution scheme. To create an expression-based distribution scheme, you use the FRAGMENT BY EXPRESSION clause of the CREATE TABLE statement. Suppose that your **my_orders** table has 30,000 rows, and you wish to distribute rows evenly across three fragments stored in **dbspace1**, **dbspace2**, and **dbspace3**. You decide to use the column **order_num** to define the expression fragments.

You can define the expression as the following example shows:

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

♦

AD/XP

If the **my_orders** table resides in a Dynamic Server with AD and XP Options database, you can create the table with a system-defined hash distribution scheme to get even distribution across fragments. Suppose that the **my_orders** table has 120,000 rows, and you wish to distribute rows evenly across six fragments stored in different dbspaces**.** You decide to use the SERIAL column **order_num** to define the fragments.

The following example shows how to use the FRAGMENT BY HASH clause to define the table with a system-defined hash distribution scheme:

```
CREATE TABLE my_orders (order_num SERIAL, ...)
    FRAGMENT BY HASH (order_num) IN

        dbspace1,
        dbspace2,
        dbspace3,
        dbspace4,
        dbspace5,
        dbspace6
```

You might notice a difference between SERIAL column values in a fragmented table and unfragmented tables. Dynamic Server with AD and XP Options assigns SERIAL values sequentially within fragments, but fragments might contain values from noncontiguous ranges. You cannot specify what these ranges are. Dynamic Server with AD and XP Options controls these ranges and guarantees only that they do not overlap.

Tip: *With Dynamic Server with AD and XP Options, you can store table fragments in dbspaces or dbslices.* ♦

Creating a Fragmented Table from Nonfragmented Tables

You might need 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.

- You have an existing large table that you want to fragment.

  Follow the instructions in the section "Creating a Fragmented Table from a Single Nonfragmented Table" on page 5-29.

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

Creating a Table from 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 the 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 the 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 expression on the common column **acc_num**.

You want rows with **acc_num** less than or equal to 1120 to be stored in the fragment that is 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 *Performance Guide*.

Creating a Fragmented Table from 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.

Modifying a Fragmented Table

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 same SQL 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 modify a fragmentation strategy with SQL statements.

Modifying Fragmentation Strategies

At times, you might need to alter a fragmentation strategy after you implement fragmentation. Most frequently, you will need 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.

AD/XP

Dynamic Server with AD and XP Options supports the ATTACH, DETACH, and INIT options of the ALTER FRAGMENT ON TABLE statement. (Tables that use HASH fragmentation support only the INIT option.) Dynamic Server with AD and XP Options does not support the ADD, DROP, and MODIFY options; however, to handle add, drop, or modify operations, you can use the supported options in place of ADD, DROP, and MODIFY. For information on the syntax of these supported options, see the ALTER FRAGMENT statement in the *Informix Guide to SQL: Syntax*.

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

Important: *Dynamic Server with AD and XP Options does not support the ALTER FRAGMENT ON INDEX statement or explicit ROWIDS columns.* ♦

Using the INIT Clause to Reinitialize a Fragmentation Scheme Completely

You can use the INIT clause when you want to reinitialize a fragmentation strategy completely. For example, suppose that you initially created the fragmented table with the following CREATE TABLE statement:

```
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 containing **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**. 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.

AD/XP

With Dynamic Server with AD and XP Options, data movement only occurs when you use an INIT clause. The database server creates a copy of the table with the new fragmentation scheme and inserts rows from the original table into the new table. ♦

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*.

IDS

Using the MODIFY Clause to Modify an Existing Fragmentation Strategy for Dynamic Server

With Dynamic Server, 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.

AD/XP

Using ATTACH and DETACH to Modify an Existing Fragmentation Strategy for Dynamic Server with AD and XP OPtions

With Dynamic Server with AD and XP Options, use the ALTER FRAGMENT statement to modify an existing fragmentation strategy. If you do not need to move data, you can use ALTER FRAGMENT statements with the following options to modify the expression of an existing fragment:

- Use the DETACH clause to remove the fragment whose expression you want to modify
- Use the ATTACH clause to reattach the fragment with the new expression.

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
```

The following statements modify the fragment that **dbspace1** contains to ensure that no account numbers with a value less than or equal to zero are stored in the fragment:

```
ALTER FRAGMENT ON TABLE account
    DETACH dbspace1 det_tab;

CREATE TABLE new_tab (acc_num INT, ...)
    FRAGMENT BY EXPRESSION
        acc_num > 0 AND acc_num <=1120 IN dbspace1;

ALTER FRAGMENT ON TABLE account
    ATTACH account, new_tab

INSERT INTO account SELECT * FROM det_tab;

DROP TABLE det_tab;
```

Tip: *If you need to move data to modify an existing fragmentation strategy, use the INIT clause of the ALTER FRAGMENT statement.*

Adding a New Fragment

When you define a fragmentation strategy, you might need to add one or more fragments. Dynamic Server with AD and XP Options and Dynamic Server use different options of the ALTER FRAGMENT statement to add a new fragment to a table. The following sections show how you might add a fragment to a table.

IDS

Using the ADD Clause to Add a Fragment for Dynamic Server

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

AD/XP

Using the ATTACH Clause to Add a Fragment for Dynamic Server with AD and XP Options

When you do not require data movement among fragments, you can use the ATTACH 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:

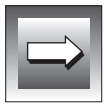
```
CREATE TABLE sales (acc_num INT, ...)
    FRAGMENT BY ROUND ROBIN IN dbspace1, dbspace2, dbspace3
```

To add a new fragment **dbspace4** to the **sales** table, you first create a new table with a structure identical to **sales** that specifies the new fragment. You then use an ATTACH clause with the ALTER FRAGMENT statement to add the new fragment to the table. The following statements add a new fragment to the **sales** table.

```
CREATE TABLE new_tab (acc_num INT, ...)
    IN dbspace4;

ALTER FRAGMENT ON TABLE sales
    ATTACH sales, new_tab
```

After you execute the ATTACH clause, the database server fragments the **sales** table into four dbspaces: the three dbspaces of **sales** and the dbspace of **new_tab**. The **new_tab** table is consumed.



Important: *You cannot use the ALTER TABLE statement with an ATTACH clause when the table has hash fragmentation. However, you can use the ALTER TABLE statement with an INIT clause on tables with hash fragmentation.*

For more information about how you can use the ATTACH clause, see the ALTER FRAGMENT statement in the *Informix Guide to SQL: Syntax*.

Dropping a Fragment

When you define a fragmentation strategy, you might need to drop one or more fragments. Dynamic Server with AD and XP Options and Dynamic Server use different options of the ALTER FRAGMENT statement to drop a fragment from a table. The following sections show how you might drop a fragment to a table.

IDS

Using the DROP Clause to Drop a Fragment

With Dynamic Server you can use the DROP clause of the ALTER FRAGMENT ON TABLE statement to drop a fragment from a table. Suppose you wish 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**.

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

AD/XP

Using the DETACH Clause to Drop a Fragment

With Dynamic Server with AD and XP Options you can use the DETACH clause of the ALTER FRAGMENT ON TABLE statement to drop a fragment from a table.

Suppose that you want to drop a fragment from a table that you create with the following statement:

```
CREATE TABLE sales (acc_num INT)...)
    FRAGMENT BY EXPRESSION
        acc_num <= 1120 IN dbspace1,
        acc_num > 1120 AND acc_num <= 2000 IN dbspace2,
        acc_num > 2000 AND acc_num < 3000 IN dbspace3,
        REMAINDER IN dbspace4
```

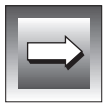
To drop the third fragment **dbspace3** from the **sales** table without losing any data, execute the following statements:

```
ALTER FRAGMENT ON TABLE sales
    DETACH dbspace3 det_tab;

INSERT INTO sales SELECT * FROM det_tab;

DROP TABLE det_tab;
```

The ALTER FRAGMENT statement detaches **dbspace3** from the distribution scheme of the **sales** table and places the rows in a new table **det_tab**. The INSERT statement reinserts rows previously in **dbspace3** into the new **sales** table, which now has three fragments: **dbspace1**, **dbspace2**, and **dbspace4**. The DROP TABLE statement drops the **det_tab** table because it is no longer needed.



Important: *You cannot use the ALTER TABLE statement with a DETACH clause when the table has hash fragmentation. However, you can use the ALTER TABLE statement with an INIT clause on tables with hash fragmentation.*

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

Fragmenting Temporary Tables

You can fragment a temporary table when you create the table. The database server deletes the fragments that are created for a temporary table at the same time it deletes the table. One restriction for fragmenting temporary tables is that you cannot alter the fragmentation strategy of a temporary table (as you can with permanent tables).

IDS

Dynamic Server allows you to create a temporary, fragmented table with the TEMP TABLE clause of the CREATE TABLE statement. ♦

AD/XP

Fragmenting Temporary Tables for Dynamic Server with AD and XP Options

Dynamic Server with AD and XP Options allows you to fragment an explicit temporary table across disks that belong to different coservers. An *explicit* temporary table is a temporary table that you create with one of the following SQL statements:

- TEMP TABLE or SCRATCH TABLE option of the CREATE TABLE statement
- INTO TEMP or INTO SCRATCH clause of the SELECT statement.

Important: *A coserver can use and access only its own dbspaces for temporary space. Although temporary tables can be fragmented explicitly across dbspaces like permanent tables, a coserver inserts data only into the fragments that it manages.*

You can define your own fragmentation strategy for an explicit temporary table or you can let Dynamic Server with AD and XP Options dynamically determine the fragmentation strategy. For information, see your *Performance Guide*.

Defining Your Own Fragmentation Strategy

To create a temporary table, use the TEMP TABLE or SCRATCH TABLE clause of the CREATE TABLE statement. To fragment a TEMP or SCRATCH table, use a FRAGMENT BY clause to specify the distribution scheme and which dbspaces or dbslices to use for the temporary table.

Letting Dynamic Server with AD and XP Options Define a Fragmentation Strategy

Dynamic Server with AD and XP Options creates and determines a fragmentation strategy while the following type of query executes:

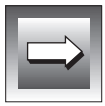
```
SELECT * FROM customer INTO SCRATCH temp_table
```

or

```
SELECT * FROM customer INTO TEMP temp_table
```

The explicit temporary table created in response to the preceding query is called a *flex (flexible) temporary table.* A flex temporary table is fragmented with a round-robin distribution scheme. You do not need to know the column names and data types for flex temporary tables, as you do with explicit temporary tables.

When you use SELECT...INTO TEMP syntax, Dynamic Server with AD and XP Options uses a flex temporary table operator to optimize how you use dbspaces and dbslices to store temporary tables. These flex operators execute the insert into these fragments in parallel (even if PDQ_PRIORITY is 0). The parallelization of the query or segment determines the number of instances of the flex SQL insert operator. Each flex temporary table operator executes on a coserver that Dynamic Server with AD and XP Options chooses as a candidate to store a fragment of the flex temporary table.



Important: *When the flexible temporary table feature stores and retrieves data from multiple dbspaces, it does so with SQL operators on the various coservers that manage those dbspaces. A coserver can use and access only its own dbspaces.*

When each flex operator receives data for a flex temporary table, Dynamic Server with AD and XP Options creates a fragment in one of the dbspaces available (based on the value of DBSPACETEMP) and light-appends the data to the fragment. If the dbspace becomes full, Dynamic Server with AD and XP Options creates a fragment of the temporary table in another dbspace, and the SQL operator continues to append data to the temporary table. If an instance of the flex insert operator does not receive any data, it does not create any fragments.

Fragmentation of Table Indexes

You can fragment both table data and table indexes. The fragmentation strategy for an index can be the same as the table-data fragmentation strategy (attached index) or independent of the table-data strategy (detached index).

Attached Indexes

An *attached index* is an index that you create without an explicit storage option. With an attached index, the number of index fragments is identical to the number of data fragments. When you create an index on a fragmented table but do not specify a distribution scheme for that index, by default, the database server fragments the index according to the same distribution scheme as the table. More specifically, the database server distributes the index keys into fragments with the same rule as the table data and places the index keys in the same dbspaces as the corresponding table data.

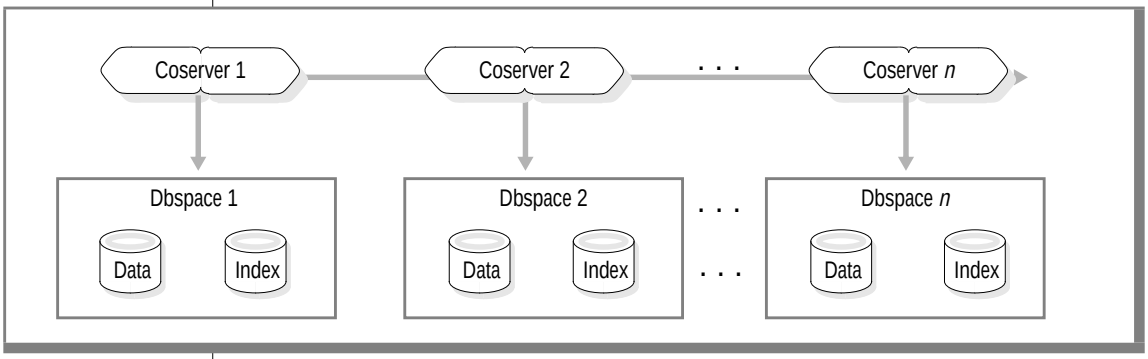
IDS

With Dynamic Server, an attached index for a nonfragmented table is stored in the same tablespace as table data, so index pages are interleaved with data pages. However, an attached index for a fragmented table is stored in a different tablespace than table data, so indexes and table data share the same dbspace but index pages are not interleaved with data pages. ♦

AD/XP

With Dynamic Server with AD and XP Options, an attached index (for both fragmented and nonfragmented tables) is stored in a different tablespace than table data, so indexes and table data share the same dbspace but index pages are not interleaved with data pages. Figure 5-11 on page 5-40 shows a storage scheme for the indexes that are attached to a fragmented table for Dynamic Server with AD and XP Options.

Figure 5-11
Storage Scheme for the Indexes Attached to a Fragmented Table for Dynamic Server with AD and XP Options



♦

Detached Indexes

The fragmentation scheme for an index can differ from that of table data. A *detached index* is an index that has a fragmentation strategy independent of the table fragmentation. You can use the FRAGMENT BY clause of the CREATE INDEX statement to fragment the index for any table.

IDS

With Dynamic Server, you can use the expression-based distribution scheme to create a detached index for any table. However, you cannot use the round-robin distribution scheme for an index.

You can change the fragmentation strategy of a fragmented index with the ALTER FRAGMENT ON INDEX statement. ♦

AD/XP

With Dynamic Server with AD and XP Options, you can use an expression, system-defined hash, or hybrid distribution scheme to create detached indexes for any table. However, you cannot use the round-robin distribution scheme for an index.

Dynamic Server with AD and XP Options does not support the ALTER FRAGMENT ON INDEX statement. To change the fragmentation strategy of a fragmented index, you must first drop the index with the DROP INDEX statement, and use the CREATE INDEX statement to recreate the index with the new fragmentation strategy. ♦

If you do not want the index on a fragmented table to be fragmented, you can place the index in a separate dbspace with the CREATE INDEX...IN DBSPACE statement.

For more information on index fragmentation, see your *Performance Guide* and the *Informix Guide to SQL: Syntax*.

Rowids

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. The recommended method to reference a row is to use the primary-key value. Primary keys are defined in the ANSI specification of SQL. Primary keys make applications more portable.

IDS

To accommodate applications that must reference a rowid for a fragmented table, Dynamic Server allows you to explicitly create a rowid column for a fragmented table. To access a row with an explicitly created rowid column is slower than with a primary key.

Creating a Rowid Column

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

You cannot create the rowid column by naming it as one of the columns in a table that you create or alter.

What Happens When You Create a Rowid Column?

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** catalog table for the internal index ♦

Accessing Data Stored in Fragmented Tables

You can use several methods to access rows that are stored in nonfragmented tables. One method is to reference the rowid of the row that you want to access.

IDS

With Dynamic Server, if you want to reference by rowid the rows that are stored in a fragmented table, you must explicitly create a rowid column. For information on creating a rowid column, see "Creating a Rowid Column in a Fragmented Table" on page 5-43. If user applications attempt to reference a rowid in a fragmented table that does not contain a rowid that you explicitly created, the database server displays an appropriate error message, and execution of the application halts. ♦

AD/XP

Dynamic Server with AD and XP Options does not support rowids on fragmented tables. You must use column values in a row to identify rows. ♦

Using Primary Keys Instead of Rowids

Informix recommends that you use primary keys rather than rowids as a method of access in your applications. Because the ANSI specification of SQL defines primary keys, when you use them to access data they make your applications more portable.

For complete information about how to define and use primary keys to access data, see the *Informix Guide to SQL: Reference* and *Informix Guide to SQL: Syntax*.

IDS

Rowid in a Fragmented Table for Dynamic Server

From the viewpoint of an application, the functionality of a rowid column in a fragmented table is identical to that of a rowid in a nonfragmented table. However, unlike the rowid of a nonfragmented table, the database server uses an index to map the rowid to a physical location. Using a rowid to access data in a fragmented table is significantly slower than using a rowid to access data in a nonfragmented table. Using a rowid to access data in a fragmented table is no faster than using a primary key to access data. In addition, primary-key access can lead to significantly improved performance in many situations, particularly when access is in parallel.

When the database server uses the rowid column to access a row in a fragmented table, it uses an index to look up the physical address of the row before it attempts to access the row. For a nonfragmented table, the database server uses direct physical access without having to perform an index lookup. Consequently, using a rowid to access a row in a fragmented table takes slightly longer than using a rowid to access a row in a nonfragmented table. You should also expect a small effect on performance when you process inserts and deletes because of the cost of maintaining the rowid index for fragmented tables.

Creating a Rowid Column in a Fragmented Table

If, for some reason, you find that your applications must use a rowid column to access data in a fragmented table, you must create a rowid column for the fragmented table.

You can use the WITH ROWIDS clause of the CREATE TABLE statement to create the rowid column at the same time that you create the table. When you issue the CREATE TABLE...WITH ROWIDS statement, the database server creates a rowid column that adds 4 bytes to each row in the fragmented table. In addition, the database server creates an internal index on the rowid column that it uses to access the data in the table. After the rowid column is created, the database server inserts a row in the **sysfragments** catalog table, which indicates the existence and attributes of the rowid column.

If you decide that you need a rowid column after you build the fragmented table, use the ADD ROWIDS clause of the ALTER TABLE statement or the INIT clause of the ALTER FRAGMENT statement.

You can drop the rowid column from a fragmented table with the DROP ROWIDS clause of the ALTER TABLE statement. For more information, see the ALTER TABLE statement in the *Informix Guide to SQL: Syntax*.

You cannot create or add a rowid column by naming it as one of the columns in a table that you create or alter. For example, you will receive an error if you execute the following statement:

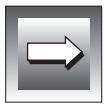
```
CREATE TABLE test_table (rowid INTEGER, ....)
```

You will get the following error:

```
-227 DDL options on rowid are prohibited. error.
```

Granting and Revoking Privileges from Fragments

You need a strategy to control data distribution if you want to grant useful fragment privileges. Fragmenting data records by expression is such a strategy. The round-robin data-record distribution strategy, on the other hand, is not a useful strategy because each new data record is added to the next fragment. This 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 are fragmented by an expression strategy.



Important: *If you issue a GRANT FRAGMENT statement or a REVOKE FRAGMENT statement against a table that is fragmented with a round-robin strategy, the command fails, and an error message is returned.*

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 at once, 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*.

AD/XP

Dynamic Server with AD and XP Options does not support the GRANT FRAGMENT and REVOKE FRAGMENT statements. ♦

Data Warehousing

Section II

Chapter

6

Building a Dimensional Data Model

Overview of Data Warehousing 6-4
Why Build a Dimensional Database? 6-5
What is Dimensional Data?. 6-6

Concepts of Dimensional Data Modeling 6-9
The Fact Table 6-10
Dimensions of the Data Model 6-11
Dimension Elements 6-12
Dimension Attributes 6-13
Dimension Tables. 6-14

Building a Dimensional Data Model 6-15
Choosing a Business Process 6-16
Summary of a Business Process 6-16
Determining the Granularity of the Fact Table 6-18
How Granularity Affects the Size of the Database 6-18
Using the Business Process to Determine the Granularity . . . 6-18
Identifying the Dimensions and Hierarchies. 6-20
Choosing the Measures for the Fact Table. 6-22
Using Keys to Join the Fact Table with the Dimension Tables 6-24
Resisting Normalization. 6-25
Choosing the Attributes for the Dimension Tables. 6-26

Handling Common Dimensional Data-Modeling Problems 6-28
Minimizing the Number of Attributes in a Dimension Table . . . 6-28
Handling Dimensions That Occasionally Change 6-30
Using the Snowflake Schema 6-32

This chapter introduces concepts and techniques of dimensional data modeling and shows how to build a simple dimensional data model. Chapter 7 shows how to use SQL to implement this dimensional data model. This chapter includes the following topics:

- Overview of Data Warehousing
- Concepts of Dimensional Data Modeling
- Building a Dimensional Data Model
- Handling Common Dimensional Data-Modeling Problems

A dimensional data model is harder to maintain for very large data warehouses than a relational data model. For this reason, data warehouses typically are based on a relational data model. However, a dimensional data model is particularly well-suited for building data marts (a subset of a data warehouse).

The general principals of dimensional data modeling that this chapter discusses are applicable for databases that your create with Informix Dynamic Server or Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options. Although no single factor determines which database server you should use to build a dimensional database, the assumption is that large, scalable warehouses are built with Dynamic Server with AD and XP Options while smaller warehouses, OLTP systems, and operational systems are built with Dynamic Server.

To understand the concepts of dimensional data modeling, you should have a basic understanding of SQL and relational database theory. This chapter provides only a summary of data warehousing concepts and describes a simple dimensional data model. To learn more advanced concepts and techniques of dimensional data modeling, Informix recommends that, after you read this chapter, you refer to the books listed in the introduction under "Related Reading" on page 15.

Overview of Data Warehousing

In the broadest sense of the term, a *data warehouse* has been used to refer to a database that contains very large stores of historical data. The data is stored as a series of snapshots, in which each record represents data at a specific point in time. This data snapshot allows a user to reconstruct history and to make accurate comparisons between different time periods. A data warehouse integrates and transforms the data that it retrieves before it is loaded into the warehouse. A primary advantage of a data warehouse is that it provides easy access to and analysis of vast stores of information.

Because the term data warehouse can mean different things to different people, this manual uses the umbrella terms d*ata warehousing* and *data-warehousing environment* to encompass any of the following forms that you might use to store your data:

- Data warehouse

  A database that is optimized for data retrieval. The data is not stored at the transaction level; some level of data is summarized. Unlike traditional OLTP databases, which automate day-to-day operations, a data warehouse provides a decision-support environment in which you can evaluate the performance of an *entire enterprise* over time. Typically, you use a relational data model to build a data warehouse.

- Data mart

  A subset of a data warehouse that is stored in a smaller database and that is oriented toward a specific purpose or data subject rather than for enterprise-wide strategic planning. A data mart can contain operational data, summarized data, spatial data, or metadata. Typically, you use a dimensional data model to build a data mart.

- Operational data store

  A subject-oriented system that is optimized for looking up one or two records at a time for decision making. An operational data store is a hybrid form of data warehouse that contains timely, current, integrated information. The data typically is of a higher level granularity than the transaction. You can use an operational data store for clerical, day-to-day decision making. This data can serve as the common source of data for data warehouses.

- Repository

  A repository combines multiple data sources into one normalized database. The records in a repository are updated frequently. Data is operational, not historical. You might use the repository for specific decision-support queries, depending on the specific system requirements. A repository fits the needs of a corporation that requires an integrated, enterprise-wide data source for operational processing.

Why Build a Dimensional Database?

Relational databases typically are optimized for on-line transaction processing (OLTP). OLTP systems are designed to meet the day-to-day operational needs of the business and the database performance is tuned for those operational needs. Consequently, the database can retrieve a small number of records quickly, but it can be slow if you need to retrieve a large number of records and summarize data on the fly. Some potential disadvantages of OLTP systems are as follows:

- Data may not be consistent across the business enterprise.
- Access to data can be complicated.

In contrast, a dimensional database is designed and tuned to support the analysis of business trends and projections. This type of informational processing is known as on-line analytical processing (OLAP) or decision-support processing. OLAP is also the term that database designers use to describe a dimensional approach to informational processing.

A dimensional database is optimized for data retrieval and analysis. Any new data that you load into the database is usually updated in batch, often from multiple sources. Whereas OLTP systems tend to organize data around specific processes (such as order entry), a dimensional database tends to be subject oriented and aims to answer questions such as “What products are selling well? At what time of year do products sell best? In what regions are sales weakest?”

The following table summarizes the key differences between OLTP and OLAP databases.

Relational Database (OLTP)	Dimensional Database (OLAP)
Data is atomized	Data is summarized
Data is current	Data is historical
Processes one record at a time	Processes many records at a time
Process oriented	Subject oriented
Designed for highly structured repetitive processing	Designed for highly unstructured analytical processing

Many of the problems that businesses attempt to solve with relational technology are multidimensional in nature. For example, SQL queries that create summaries of product sales by region, region sales by product, and so on, might require hours of processing on a traditional relational database. However, a dimensional database could process the same queries in a fraction of the time.

What is Dimensional Data?

Traditional relational databases are organized around a list of records. Each record contains related information that is organized into attributes (fields). The **customer** table of the **stores7** demonstration database, which includes fields for name, company, address, phone, and so forth, is a typical example. While this table has several fields of information, each row in the table pertains to only one customer. If you wanted to create a two-dimensional matrix with customer name and any other field (for example, phone number), you realize that there is only a one to one correspondence. Figure 6-1 on page 6-7 shows that a table with fields that have only a one-to-one correspondence.

Figure 6-1
A Table with a One-To-One Correspondence Between Fields

Customer	Phone number --->		
Ludwig Pauli	408-789-8075	----------	----------
Carole Sadler	----------	415-822-1289	----------
Philip Currie	----------	----------	414-328-4543

You could put any combination of fields from the preceding customer table in this matrix, but you always end up with a one-to-one correspondence, which shows that this table is not multidimensional and would not be well suited for a dimensional database.

However, consider a relational table that contains more than a one-to-one correspondence between the fields of the table. Suppose you create a table that contains sales data for products sold in each region of the country. For simplicity, suppose the company has three products which are sold in three regions. Figure 6-2 shows how you might store this data in a relational table.

Figure 6-2
A Simple Relational Table

Product	Region	Unit Sales
Football	East	2300
Football	West	4000
Football	Central	5600
Tennis racket	East	5500
Tennis racket	West	8000
Tennis racket	Central	2300
Baseball	East	10000
Baseball	West	22000
Baseball	Central	34000

The table in Figure 6-2 on page 6-7 lends itself to multidimensional representation because it has more than one product per region and more than one region per product. Figure 6-3 shows a two-dimensional matrix that better represents the many-to-many relationship of product and region data.

Product ↓ / Region →	central	east	west
Football	5600	2300	4000
Tennis Racket	2300	5500	8000
Baseball	34000	10000	22000

Figure 6-3
A Simple Two-Dimensional Example

Although this data can be forced into the three-field relational table of Figure 6-2, the data fits more naturally into the two-dimensional matrix of Figure 6-3.

The performance advantages of the *dimensional* table over the traditional relational table can be great. A dimensional approach simplifies access to the data that you want to summarize or compare. For example, if you use the dimensional table to query the number of products sold in the West, the database server finds the **west** column and calculates the total for all row values in that column. To perform the same query on the relational table, the database server has to search and retrieve each row where the **region** column equals `west` and then aggregate the data. In queries of this kind, the dimensional table can total all values of the **west** column in a fraction of the time it takes the relational table to find all the `west` records.

Concepts of Dimensional Data Modeling

To build a dimensional database, you start with a dimensional data model. The dimensional data model provides a method for making databases simple and understandable. You can conceive of a dimensional database as a database *cube* of three or four dimensions where users can access a slice of the database along any of its dimensions. To create a dimensional database, you need a model that lets you visualize the data.

Suppose your business sells products in different markets and evaluates the performance over time. It is easy to conceive of this business process as a cube of data, which contains dimensions for time, products, and markets. Figure 6-4 shows this dimensional model. The various intersections along the lines of the cube would contain the *measures* of the business. The measures correspond to a particular combination of product, market, and time data.

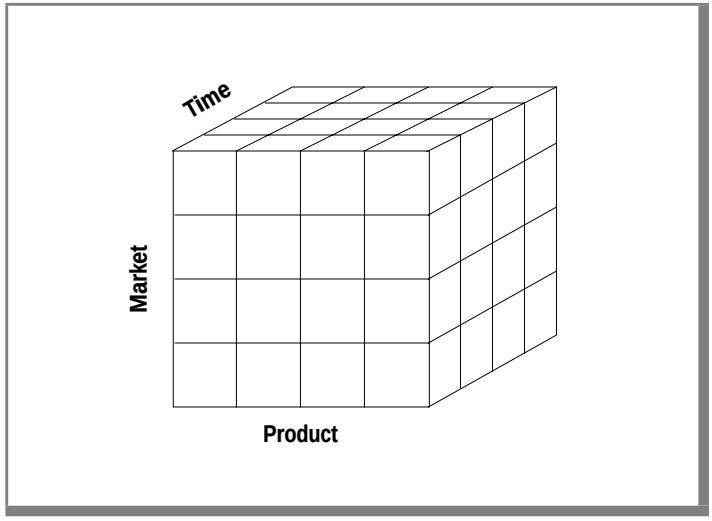


Figure 6-4 *A Dimensional Model of a Business That Has Time, Product, and Market Dimensions*

Another name for the dimensional model is the *star-join schema*. The database designers use this name because the diagram for this model looks like a star with one central table around which a set of other tables are displayed. The central table is the only table in the schema with multiple joins connecting it to all the other tables. This central table is called the *fact table* and the other tables are called *dimension tables*. The dimension tables all have only a single join that attaches them to the fact table, regardless of the query. Figure 6-5 shows a simple dimensional model of a business that sells products in different markets and evaluates business performance over time.

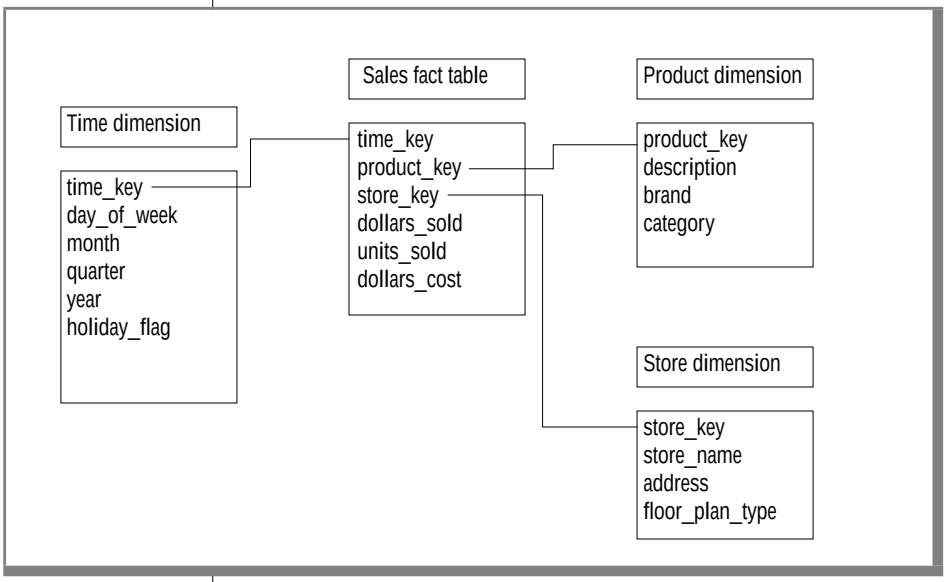


Figure 6-5
A Typical Dimensional Model

The Fact Table

The fact table stores the measures of the business and points to the key value at the lowest level of each dimension table. The *measures* are quantitative or factual data about the subject. The measures are generally numeric and correspond to the *how much* or *how many* aspects of a question. Examples of measures are price, product sales, product inventory, revenue, and so forth. A measure can be based on a column in a table or it can be calculated.

Figure 6-6 shows a fact table whose measures are sums of the units sold, the revenue, and the profit for the sales of that product to that account on that day.

Product Code	Account Code	Day Code	Units Sold	Revenue	Profit
1	5	32104	1	82.12	27.12
3	17	33111	2	171.12	66.00
1	13	32567	1	82.12	27.12

Figure 6-6
A Fact Table with Sample Records

Before you design a fact table, you must determine the *granularity* of the fact table. The granularity corresponds to how you define an individual low-level record in that fact table. The granularity might be the individual transaction, a daily snapshot, or a monthly snapshot. The fact table of Figure 6-6 contains one row for every product sold to each account each day. Thus, the granularity of the fact table is expressed as *product by account by day.*

Dimensions of the Data Model

A *dimension* represents a single set of objects or events in the real world. Each dimension that you identify for the data model gets implemented as a dimension table. Dimensions are the qualifiers that make the measures of the fact table meaningful because they answer the what, when, and where aspects of a question. For example, consider the following business questions, for which the dimensions are italicized:

- What *accounts* produced the highest revenue last *year*?
- What was our profit by *vendor*?
- How many units were sold for each *product*?

In the preceding set of questions revenue, profit, and units sold are measures, (*not* dimensions) as each represents quantitative or factual data.

Dimension Elements

A dimension can define multiple *dimension elements* for different levels of summarization. For example, all of the elements that relate to the structure of a sales organization might comprise one dimension. Figure 6-7 shows the dimension elements that the accounts dimension defines.

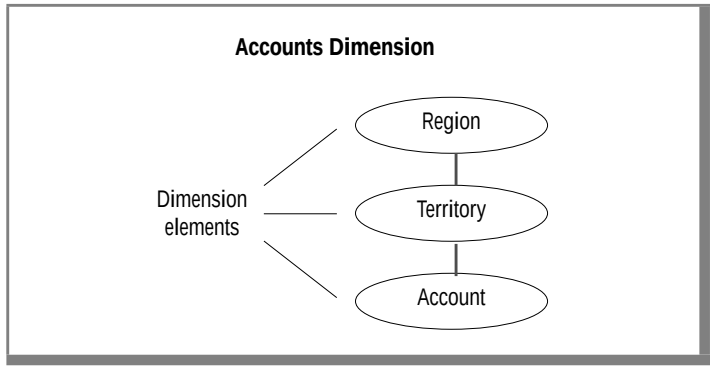


Figure 6-7 *Dimension Elements in the Accounts Dimension*

Dimensions are made up of hierarchies of related elements. Because of the hierarchical aspect of dimensions, users are able to construct queries that access data at a higher level (*roll up*) or lower level (*drill down*) than the previous level of detail. Figure 6-7 shows the hierarchical relationships of the dimension elements: accounts roll up to territories, and territories roll up to regions. Users can query at different levels of the dimension, depending on the data they want to retrieve. For example, users might perform a query against all regions and then drill down to the territory or account level for more detailed information.

Dimension elements are usually stored in the database as numeric codes or short character strings to facilitate joins to other tables.

Each dimension element can define multiple dimension attributes, in the same way dimensions can define multiple dimension elements.

Dimension Attributes

A dimension attribute is a column in a dimension table. Each attribute describes a level of summarization within a dimension hierarchy. The dimension elements define the hierarchical relationships within a dimension table; the attributes describe dimension elements in terms that are familiar to users. Figure 6-8 shows the dimension elements and corresponding attributes of the account dimension.

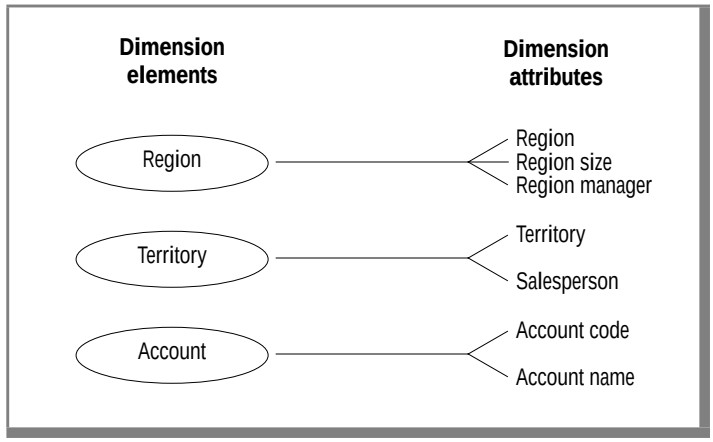


Figure 6-8 *Attributes that Correspond to the Dimension Elements*

Since dimension attributes describe the items in a dimension, they are most useful when they are text.

Tip: *Sometimes during the design process, it is unclear whether a numeric data field from a production data source is a measured fact or an attribute. Generally, if the numeric data field is a measurement that changes each time we sample it, it is a fact. If it is a discretely valued description of something that is more or less constant, it is a dimension attribute.*

Dimension Tables

A dimension table is a table that a stores the textual descriptions of the dimensions of the business. A dimension table contains an element and an attribute, if appropriate, for each level in the hierarchy. The lowest level of detail that is required for data analysis determines the lowest level in the hierarchy. Levels higher than this base level store redundant data. This denormalized table reduces the number of joins that are required for a query and makes it easier for users to query at higher levels and then drill down to lower levels of detail. The term *drilling down* means to add row headers from the dimension tables to your query. Figure 6-9 shows an example of a dimension table that is based on the account dimension.

Acct Code	Account Name	Territory	Salesman	Region	Region Size	Region Manager
1	Jane's Mfg.	101	B. Adams	Midwest	Over 50	T. Sent
2	TBD Sales	101	B. Adams	Midwest	Over 50	T. Sent
3	Molly's Wares	101	B. Adams	Midwest	Over 50	T. Sent
4	The Golf Co.	201	T. Scott	Midwest	Over 50	T. Sent

Figure 6-9
An Example of a Dimension Table

Building a Dimensional Data Model

To build a dimensional data model, you need a methodology that outlines the decisions you need to make to complete the database design. This methodology uses a top-down approach because it first identifies the major process(es) in your organization where data is collected. An important task of the database designer is to start with the existing sources of data that your organization uses. Once the process(es) are identified, one or more fact tables are built from each business process. The following steps describe the methodology you use to build the data model.

To build a dimensional database

1. Choose the business process(es) that you want to use to analyze the subject area to be modeled
2. Determine the granularity of the fact table(s)
3. Identify dimensions and hierarchies for each fact table
4. Identify measures for the fact table(s)
5. Determine the attributes for each dimension table
6. Get users to verify the data model

Although a dimensional database can be based on multiple business processes and can contain many fact tables, the data model that this section describes is based on a single business process and has one fact table.

Choosing a Business Process

A *business process* is an important operation in your organization that some legacy system supports. You collect data from this system to use in your dimensional database. The business process identifies what end users are doing with their data, where the data comes from, and how to transform that data to make it meaningful. The information can come from many sources, including finance, sales analysis, market analysis, customer profiles. The following list shows different business processes you might use to determine what data to include in your dimensional database:

- Sales
- Shipments
- Inventory
- Orders
- Invoices

Summary of a Business Process

Suppose your organization wants to analyze customer buying trends by product line and region so that you can develop more effective marketing strategies. In this scenario, the subject area for your data model is sales.

After many interviews and thorough analysis of your sales business process, suppose your organization collects the following information:

- Customer-base information has changed.

  Previously, sales districts were divided by city. Now the customer base corresponds to two regions: Region 1 for California and Region 2 for all other states.
- The following reports are most critical to marketing:
  - Monthly revenue, cost, net profit by product line per vendor
  - Revenue and units sold by product, by region, by month
  - Monthly customer revenue
  - Quarterly revenue per vendor
- Most sales analysis is based on monthly results but you may choose to analyze sales by week or accounting period (at a later date).

- A data-entry system exists in a relational database.

  To develop a working data model, you can assume that the relational database of sales information has the following properties:

  - The **stores7** database provides much of the revenue data that the marketing department uses.
  - The product code that analysts use is stored in the **catalog** table as the catalog number.
  - The product line code is stored in the **stock** table as the stock number. The product line name is stored as description.
  - The product hierarchies are somewhat complicated. Each product line has many products and each manufacturer has many products.

- All the cost data for each product is stored in a flat file named **costs.lst** on a different purchasing system.
- Customer data is stored in the **stores7** database.

  The region information has not yet been added to the database.

An important characteristic of the dimensional model is that it uses business labels familiar to end users rather than internal table or column names. Once the business process is completed, you should have all the information you need to create the measures, dimensions, and relationships for the dimensional data model. This dimensional data model is used to implement the **sales_demo** database that Chapter 7 describes.

The **stores7** demonstration database is the primary data source for the dimensional data model that this chapter develops. For detailed information about the data sources that are used to populate the tables of the **sales_demo** database, see "Mapping Data from Data Sources to the Database" on page 7-6.

Determining the Granularity of the Fact Table

Once you have gathered all the relevant information about the subject area, the next step in the design process is to determine the granularity of the fact table. To do this you must decide what an individual low-level record in the fact table should contain. The components that make up the granularity of the fact table correspond directly with the dimensions of the data model. Thus, when you define the granularity of the fact table, you identify the dimensions of the data model.

How Granularity Affects the Size of the Database

The granularity of the fact table also determines how much storage space the database requires. For example, consider the following possible granularities for a fact table:

- Product by day by region
- Product by month by region

The size of a database that has a granularity of *product by day by region* would be much greater than a database with a granularity of *product by month by region* because the database contains records for every transaction made each day as opposed to a monthly summation of the transactions. You must carefully determine the granularity of your fact table because too fine a granularity could result in an astronomically large database. Conversely, too coarse a granularity could mean the data is not detailed enough for users to perform meaningful queries against the database.

Using the Business Process to Determine the Granularity

A careful review of the information gathered from the business process should provide what you need to determine the granularity of the fact table. To summarize, your organization wants to analyze customer buying trends by product line and region so that you can develop more effective marketing strategies.

Customer by Product

The granularity of the fact table always represents the lowest level for each corresponding dimension. When you review the information from the business process, the granularity for customer and product dimensions of the fact table are apparent. Customer and product cannot be reasonably reduced any further: they already express the lowest level of an individual record for the fact table. (In some cases, product might be further reduced to the level of *product componen*t since a product could be made up of multiple components.)

Customer by Product by District

Because the customer buying trends your organization wants to analyze include a geographical component, you still need to decide the lowest level for region information. The business process indicates that in the past, sales districts were divided by city, but now your organization distinguishes between two regions for the customer base: Region 1 for California and Region 2 for all other states. Nonetheless, at the lowest level, your organization still includes sales district data, so *district* represents the lowest level for geographical information and provides a third component to further define the granularity of the fact table.

Customer by Product by District by Day

Customer buying trends always occur over time, so the granularity of the fact table must include a time component. Suppose your organization decides to create reports by week, accounting period, month, quarter, or year. At the lowest level, you probably want to choose a base granularity of *day*. This granularity allows your business to compare sales on Tuesdays with sales on Fridays, compare sales for the first day of each month, and so forth. The granularity of the fact table is now complete.

The decision to choose a granularity of day means that each record in the **time** dimension table represents a day. In terms of the storage requirements, even 10 years of daily data is only about 3,650 records, which is a relatively small dimension table.

Identifying the Dimensions and Hierarchies

Once you determine the granularity of the fact table it is easy to identify the primary dimensions for the data model because each component that defines the granularity corresponds to a dimension. Figure 6-10 shows the relationship between the granularity of the fact table and the dimensions of the data model.

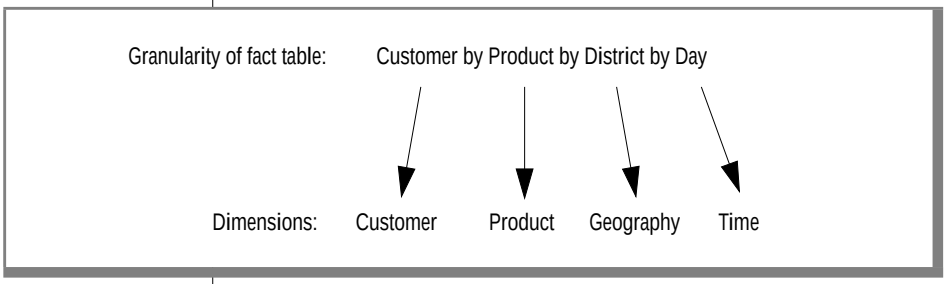


Figure 6-10 *The Granularity of the Fact Table Corresponds to the Dimensions of the Data Model*

With the dimensions (customer, product, geography, time) for the data model in place, the schema diagram begins to take shape.

Tip: *At this point, you can add additional dimensions to the primary granularity of the fact table, where the new dimensions take on only a single value under each combination of the primary dimensions. If you see that an additional dimension violates the granularity because it causes additional records to be generated, then you must revise the granularity of the fact table to accommodate the additional dimension. For this data model, no additional dimensions need to be added.*

You can now map out dimension elements and hierarchies for each dimension. Figure 6-11 shows the relationship among dimensions, dimension elements, and the inherent hierachies.

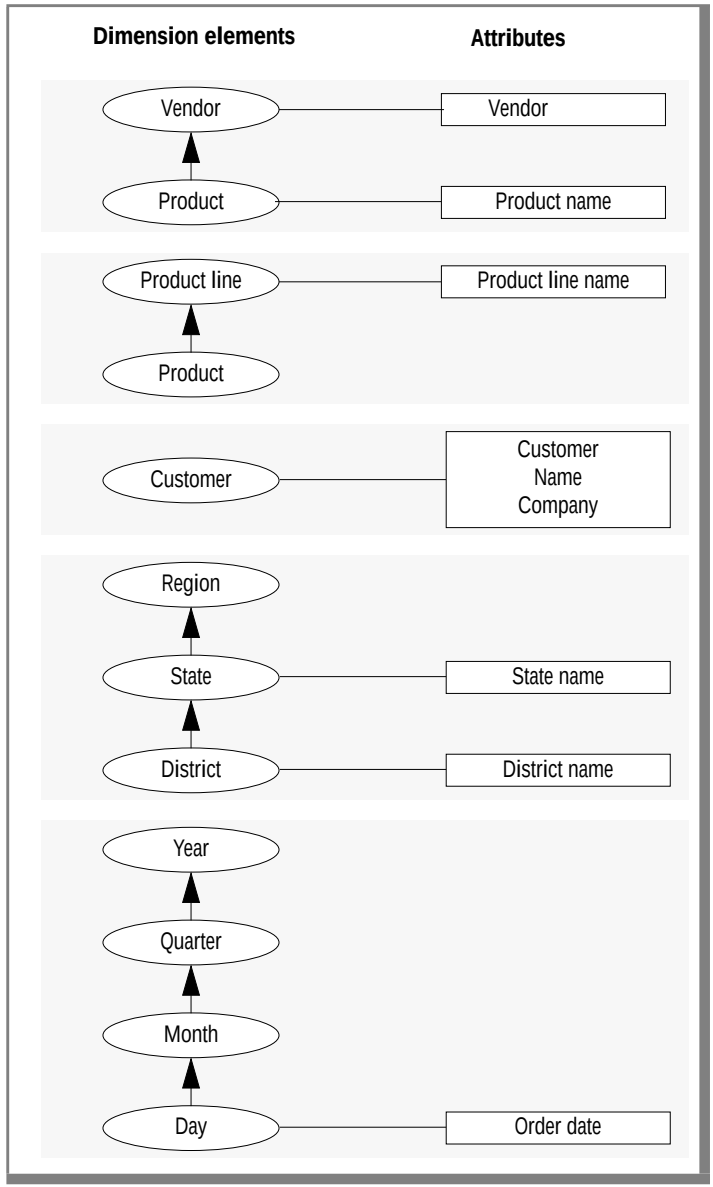


Figure 6-11 *The Relationships Between Dimensions and Dimension Elements and the Inherent Hierarchies*

In most cases, the dimension elements need to express the lowest possible granularity for each dimension, not because queries need to access individual low-level records, but because queries need to cut through the database in precise ways. In other words, even though the questions that a data warehousing environment poses are usually broad, these questions still depend on the lowest level of product detail.

Choosing the Measures for the Fact Table

The measures for the data model include not only the data itself, but also new values that you calculate from the existing data. When you examine the measures, you might discover that you need to make adjustments either in the granularity of the fact table or the number of dimensions.

Another important decision you must make when you design the data model is whether to store the calculated results in the fact table or to derive these values at runtime.

The first question to answer is "what measures are used to analyze the business?" Remember that the measures are the quantitative or factual data that tell *how much* or *how many*. The information that you gather from analysis of the sales business process results in the following list of measures:

- Revenue
- Cost
- Units sold
- Net profit

You use these measures to complete the fact table in Figure 6-12.

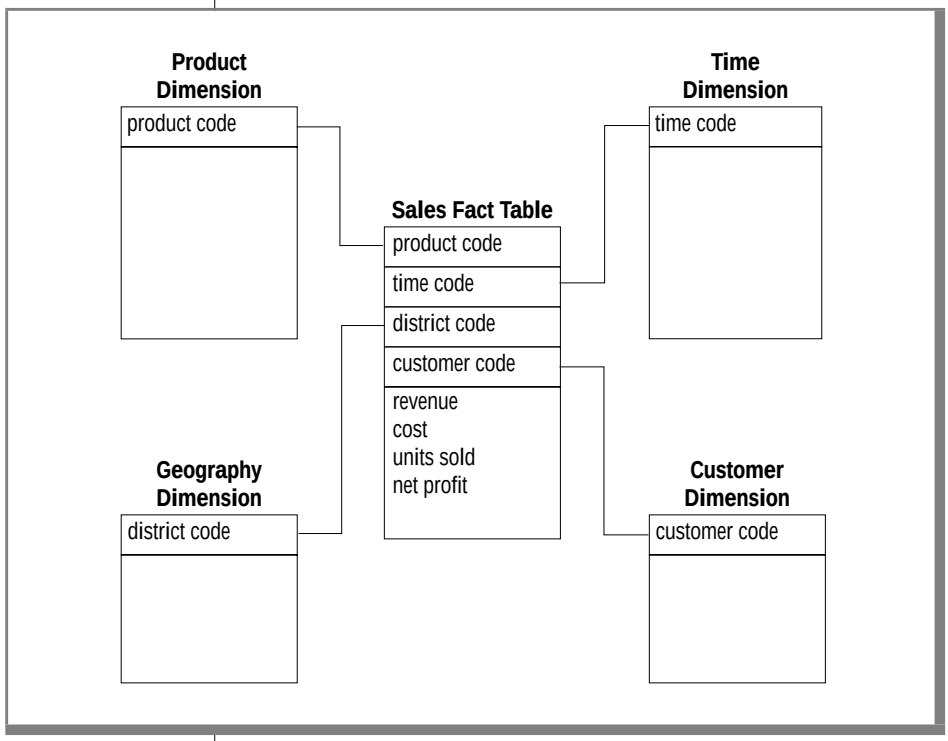


Figure 6-12
The Sales Fact Table References Each Dimension Table

Using Keys to Join the Fact Table with the Dimension Tables

Assume, for the moment, that the schema of Figure 6-12 on page 6-23 shows both the logical and physical design of the database. The database contains the following five tables:

- Sales fact table
- Product dimension table
- Time dimension table
- Customer dimension table
- Geography dimension table

Each of the dimensional tables includes a primary key (product, time_code, customer, district_code), and the corresponding columns in the fact table are foreign keys. The fact table also has a primary (composite) key that is a combination of these four foreign keys. As a rule, each foreign key of the fact table must have its counterpart in a dimension table. Furthermore, any table in a dimensional database that has a composite key must be a fact table, which means that every table in a dimensional database that expresses a many-to-many relationship is a fact table.

Tip: *The primary key should be a short numeric data type (INT, SMALLINT, SERIAL) or a short character string (as used for codes). Informix recommends that you do not use long character strings as primary keys.*

Resisting Normalization

If the four foreign keys of the fact table are tightly administered consecutive integers, you could reserve as little as 16 bytes for all four keys (4 bytes each for time, product, customer, and geography) of the fact table. If the four measures in the fact table were each 4 byte integer columns, you would need to reserve only another 16 bytes. Thus, each record of the fact table would be only 32 bytes. Even a billion-row fact table would require only about 32 gigabytes of primary data space.

With its compact keys and data, such a storage-lean fact table is typical for dimensional databases. The fact table in a dimensional model is by nature highly normalized. You cannot further normalize the extremely complex many-to-many relationships among the four keys in the fact table because no correlation exists between the four dimension tables; virtually every product is sold every day to all customers in every region.

The fact table is the largest table in a dimensional database. Because the dimension tables are usually much smaller than the fact table, you can ignore the dimension tables when you calculate the disk space for your database. Efforts to normalize any of the tables in a dimensional database solely to save disk space are pointless. Furthermore, normalized dimension tables undermine the ability of users to explore a single dimension table to set constraints and choose useful row headers.

Choosing the Attributes for the Dimension Tables

Once the fact table is complete, you can decide the dimension attributes for each of the dimension tables. To illustrate how to choose the attributes, consider the time dimension. The data model for the sales business process defines a granularity of day that corresponds to the time dimension, so that each record in the **time** dimension table represents a day. Keep in mind that each field of the table is defined by the particular day the record represents.

The analysis of the sales business process also indicates that the marketing department needs monthly, quarterly, and annual reports, so the time dimension includes the elements: day, month, quarter, and year. Each element is assigned an attribute that describes the element and a code attribute (to avoid column values that contain long character strings). Figure 6-13 shows the attributes for the **time** dimension table and sample values for each field of the table.

Figure 6-13
Attributes for the Time Dimension

time code	order date	month code	month	quarter code	quarter	year
35276	07 / 31 / 1996	7	july	3	third q	1996
35277	08 / 01 / 1996	8	aug	3	third q	1996
35278	08 / 02 / 1996	8	aug	3	third q	1996

Figure 6-13 on page 6-26 shows that the attribute names you assign should be familiar business terms that make it easy for end users to form queries on the database. Figure 6-14 shows the completed data model for the sales business process, with all the attributes defined for each dimension table.

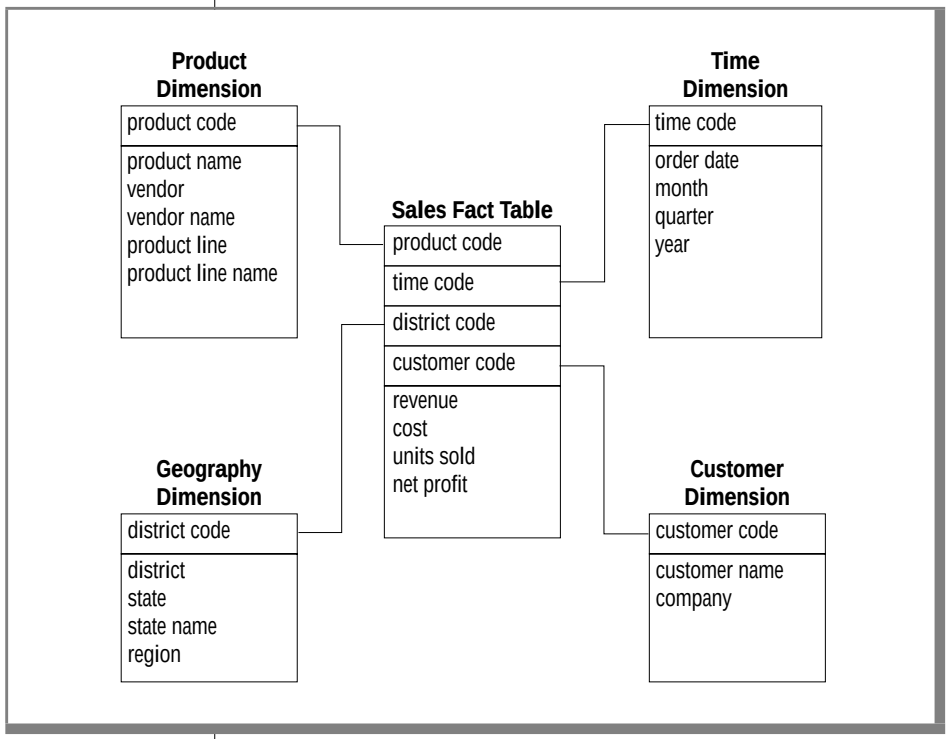


Figure 6-14 *The Completed Dimensional Data Model for the Sales Business Process*

Tip: *The number of attributes that you define on each dimension table should generally be kept to a minimum. Dimension tables with too many attributes can lead to excessively wide rows and poor performance. For more information, see "Minimizing the Number of Attributes in a Dimension Table" on page 6-28.*

Handling Common Dimensional Data-Modeling Problems

The dimensional model that the previous sections describe illustrates only the most basic concepts and techniques of dimensional data modeling. The data model you build to address the business needs of your enterprise typically involves additional problems and difficulties that you must resolve to achieve the best possible query performance from your database. This section describes various methods you can use to resolve some of the most common problems that arise when you build a dimensional data model.

Minimizing the Number of Attributes in a Dimension Table

Dimension tables that contain customer or product information might easily have 50 to 100 attributes and many millions of rows. However, dimension tables with too many attributes can lead to excessively wide rows and poor performance. For this reason, you might want to separate out certain groups of attributes from a dimension table and put them in a separate table called a *mini-dimension* table. A mini-dimension table consists of a small group of attributes that are separated out from a larger dimension table. You might choose to create a mini-dimension table for attributes that have either of the following characteristics:

- The fields are rarely used as constraints in a query.
- The fields are frequently compared together.

Figure 6-15 shows a mini-dimension table for demographic information that is separated out from a customer table.

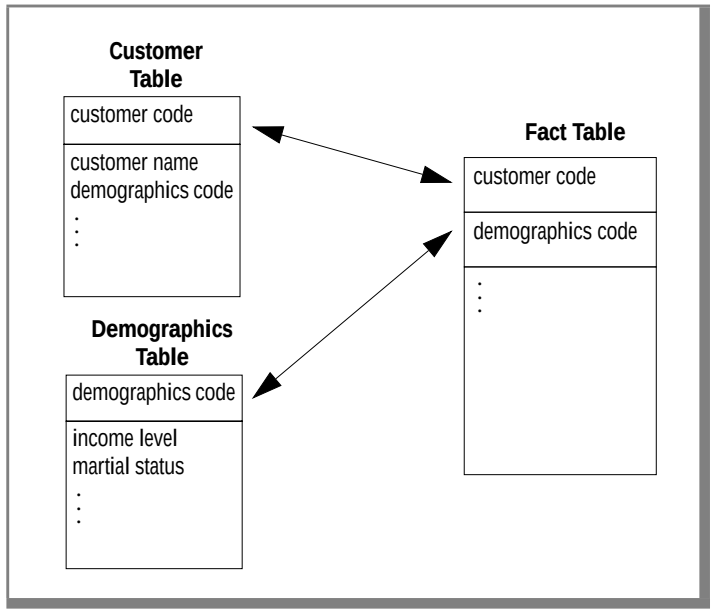


Figure 6-15 *A Mini-Dimension Table for Demographics Information*

In the demographics table, you can store the demographics key as a foreign key in both the fact table and the customer table. This allows you join the demographics table directly to the fact table. You can also use the demographics key directly with the customer table to browse demographic attributes.

Handling Dimensions That Occasionally Change

In a dimensional database where updates are infrequent (as opposed to OLTP systems), most dimensions are *relatively* constant over time, since changes in sales districts or regions, or in company names and addresses, occur infrequently. However, to make historical comparisons, these changes must be handled when they do occur. Figure 6-16 shows an example of a dimension that has changed.

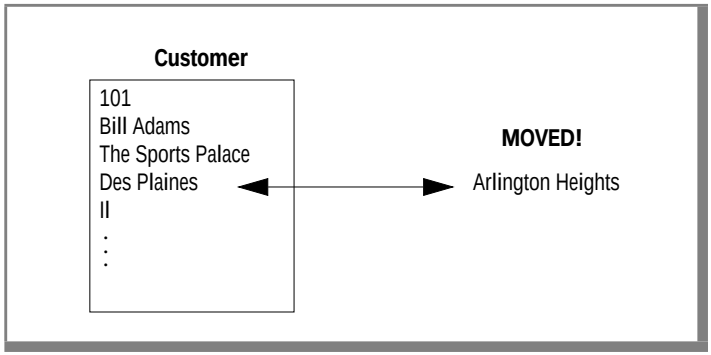


Figure 6-16
A Dimension That Changes

You can use three methods to handle changes that occur in a dimension:

- Change the value stored in the dimension column.

  In Figure 6-16 on page 6-30, the record for `Bill Adams` in the customer dimension table is updated to show the new address `Arlington Heights`. All of this customer's previous sales history is now associated with the district of Arlington Heights instead of Park Ridge.

- Create a second dimension record with the new value and a generalized key.

  This approach effectively partitions history. The customer dimension table would now contain two records for Bill Adams. The old record with a key of 101 remains, and records in the fact table are still associated with it. A new record is also added to the customer table for Bill Adams, with a new key that might consist of the old key plus some version digits (101.01, for example). All subsequent records that are added to the fact table for Bill Adams are associated with this new key.

- Add a new field in the dimension table for the affected attribute and rename the old attribute.

  This approach is rarely used unless you need to track old history in terms of the new value, and vice-versa. The customer table gets a new attribute named **current address**, and the old attribute is renamed **original address**. The record that contains information about Bill Adams includes values for both the original and current address.

Using the Snowflake Schema

A snowflake schema is a variation on the star schema, in which very large dimension tables are normalized into multiple tables. Dimensions with hierarchies can be decomposed into a snowflake structure when you want to avoid joins to big dimension tables when you are using an aggregate of the fact table. For example, if you have brand information that you want to separate out from a product dimension table, you can create a brand snowflake that consists of a single row for each brand and that contains significantly fewer rows than the product dimension table. Figure 6-17 shows a snowflake structure for the brand and product line elements and the **brand_agg** aggregate table.

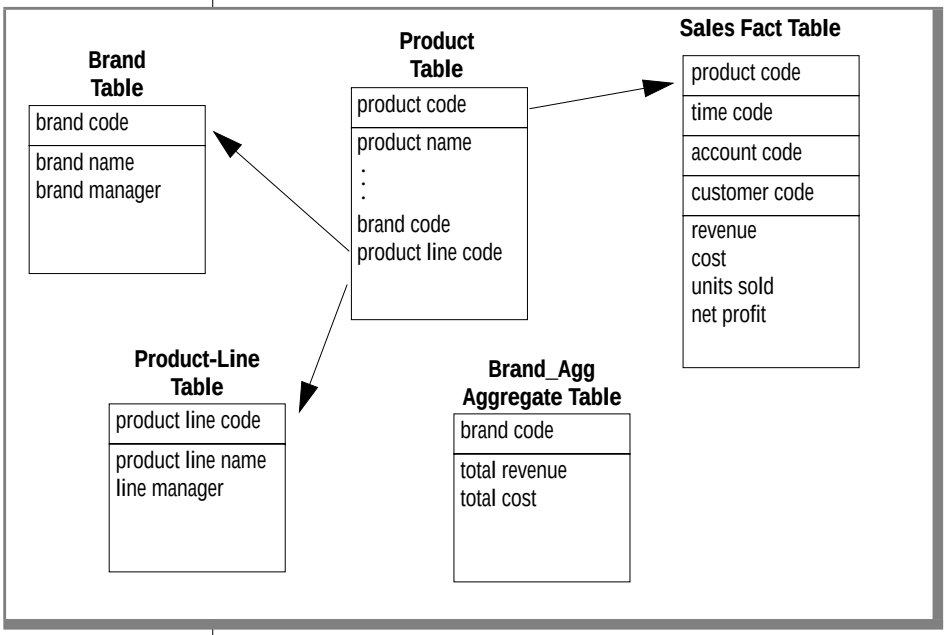


Figure 6-17
An Example of a Snowflake Schema

If you create an aggregate, **brand_agg**, that consists of the brand code and the total revenue per brand, you can use the snowflake schema to avoid the join to the much larger **sales** table, as the following query on the **brand** and **brand_agg** tables shows:

```
SELECT brand.brand_name, brand_agg.total_revenue
FROM brand, brand_agg
    WHERE brand.brand_code = brand_agg.brand_code
    AND brand.brand_name = 'Anza'
```

Without a *snowflaked* dimension table, you use a SELECT UNIQUE or SELECT DISTINCT statement on the entire **product** table (potentially, a very large dimension table that includes all the brand and product-line attributes) to eliminate duplicate rows.

While snowflake schemas are unnecessary when the dimension tables are relatively small, a retail or mail-order business that has customer or product dimension tables that contain millions of rows can use snowflake schemas to significantly improve performance.

If an aggregate table is not available, any joins to a dimension element that was normalized with a snowflake schema must now be a three-way join, as the following query shows. A three-way join reduces some of the performance advantages of a dimensional database.

```
SELECT brand.brand_name, SUM(sales.revenue)
FROM product, brand, sales
    WHERE product.brand_code = brand.brand_code
    AND brand.brand_name = 'Alltemp'
GROUP BY brand_name
```

Chapter

7

Implementing a Dimensional Data Model

Implementing the Dimensional Database 7-3
Using CREATE DATABASE 7-3
Using CREATE TABLE for the Dimension and Fact Tables 7-4
Mapping Data from Data Sources to the Database 7-6
Loading Data into the Dimensional Database 7-9
Using Command Files to Create the sales_demo Database 7-11
Testing the Dimensional Database 7-12

Logging and Nonlogging Tables for Dynamic Server with AD and XP Options 7-13
Choosing Table Types 7-14
Scratch and Temp Tables 7-15
Raw Permanent Tables 7-16
Static Permanent Tables 7-16
Operational Permanent Tables 7-17
Standard Permanent Tables 7-17
Switching Between Table Types 7-18

Indexes for Data-Warehousing Environments 7-18
Using GK Indexes in a Data-Warehousing Environment. 7-20
Defining a GK Index on a Selection 7-20
Defining a GK Index on an Expression 7-21
Defining a GK Index on Joined Tables. 7-21

This chapter shows you how to use SQL to implement the dimensional data model that Chapter 6 describes. Remember that this database serves only as an illustrative example of a data warehousing environment. For the sake of the example, it is translated into SQL statements.

This chapter also describes the table types and indexes for Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options that are suited to the needs of data warehousing and other very large database applications.

Implementing the Dimensional Database

This section shows the SQL statements you can use to create a dimensional database from the data model in Chapter 6. You can use interactive SQL to write the individual statements that create the database or you can run command scripts that automatically execute all the statements that you need to implement the database. The CREATE DATABASE and CREATE TABLE statements create the data model as tables in a database. Once you create the database you can use LOAD and INSERT statements to populate the tables.

Using CREATE DATABASE

You must create the database before you can create any tables or other objects that the database contains.

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

IDS

When you create a database with Informix Dynamic Server, you can turn logging off. ♦

AD/XP

When you create a database with Dynamic Server with AD and XP Options, logging is always turned on. The following statement creates a database with logging that is called **sales_demo**:

```
CREATE DATABASE sales_demo
```

Any database that you create with Dynamic Server with AD and XP Options is a logged database; however, you can create nonlogging tables within the database. For information on the logging and nonlogging tables that Dynamic Server with AD and XP Options supports, see "Logging and Nonlogging Tables for Dynamic Server with AD and XP Options" on page 7-13. ♦

Using CREATE TABLE for the Dimension and Fact Tables

This section includes the CREATE TABLE statements that you use to create the tables of the **sales_demo** dimensional database.

Referential integrity is, of course, an important requirement for dimensional databases. However, the following schema for the **sales_demo** database does not define the primary and foreign key relationships that exist between the fact table and its dimension tables. The schema does not define these primary and foreign key relationships because data-loading performance improves dramatically when the database server does not enforce constraint checking. Given that data warehousing environments often require that tens or hundreds of gigabytes of data are loaded within a specified time, data-load performance should be a factor when you decide how to implement a database in a warehousing environment. Assume that if the **sales_demo** database is implemented as a live data mart, some data extraction tool (rather than the database server) is used to enforce referential integrity between the fact table and dimension tables.

Tip: *After you create and load a table, you can add primary- and foreign-key constraints to the table with the ALTER TABLE statement to enforce referential integrity. This method is required only for express load mode. If the constraints and indexes are necessary and costly to drop before a load, then deluxe load mode is the best option.*

The following statements create the **time**, **geography**, **product**, and **customer** tables. These tables are the dimensions for the **sales** fact table. A SERIAL field serves as the primary key for the **district_code** column of the **geography** table.

```
CREATE TABLE time
    (
time_code INT,
order_date DATE,
month_code SMALLINT,
month_name CHAR(10),
quarter_code SMALLINT,
quarter_name CHAR(10),
year INTEGER
    );

CREATE TABLE geography (
district_code SERIAL,
district_name CHAR(15),
state_code CHAR(2),
state_name CHAR(18),
region SMALLINT
);

CREATE TABLE product (
product_code INTEGER,
product_name CHAR(31),
vendor_code CHAR(3),
vendor_name CHAR(15),
product_line_code SMALLINT,
product_line_name CHAR(15)
);

CREATE TABLE customer (
customer_code INTEGER,
customer_name CHAR(31),
company_name CHAR(20));
```

The **sales** fact table has pointers to each dimension table. For example, **customer_code** references the customer table, **district_code** references the geography table, and so forth. The **sales** table also contains the measures for the units sold, revenue, cost, and net profit.

```
CREATE TABLE sales
(
customer_code INTEGER,
district_code SMALLINT,
time_code INTEGER,
product_code INTEGER,
units_sold SMALLINT,
revenue MONEY(8,2),
cost MONEY(8,2),
net_profit MONEY(8,2)
);
```

Tip: *The most useful measures (facts) are numeric and additive. Because of the great size of databases in data warehousing environments, virtually every query against the fact table might require thousands or millions of records to construct an answer set. The only useful way to compress these records is to aggregate them. In the* ***sales*** *table, each column for the measures is defined on a numeric data type, so you can easily build answer sets from the* ***units_sold****,* ***revenue****,* ***cost****, and* ***net_profit*** *columns.*

For your convenience, the file called **createdw.sql** contains all the preceding CREATE TABLE statements.

UNIX

When your database server runs on the UNIX operating system you can access the **createdw.sql** file from the directory **$INFORMIXDIR/demo/dbaccess**. ♦

WIN NT

When your database server runs on the Windows NT operating system you can access the **createdw.sql** file from the directory **%INFORMIXDIR%\demo\dbaccess**. ♦

Mapping Data from Data Sources to the Database

The **stores7** demonstration database is the primary data source for the **sales_demo** database.

Figure 7-1 on page 7-7 shows the relationship between data-warehousing business terms and the data sources. It also shows the data source for each column and table of the **sales_demo** database.

Figure 7-1
The Relationship Between Data-Warehousing Business Terms and Data Sources

Business term	Data source	Table.Column name
Sales Fact Table:		
product code		sales.product_code
customer code		sales.customer_code
district code		sales.district_code
time code		sales.time_code
revenue	stores7:items.total_price	sales.revenue
units sold	stores7:items.quantity	sales.units_sold
cost	costs.lst (per unit)	sales.cost
net profit	calculated: revenue minus cost	sales.net_profit
Product Dimension Table:		
product	stores7:catalog.catalog_num	product.product_code
product name	stores7:stock.manu_code and stores7:stock.description	product.product_name
product line	stores7:orders.stock_num	product.product_line_code
product line name	stores7:stock.description	product.product_line_name
vendor	stores7:orders.manu_code	product.vendor_code
vendor name	stores7:manufact.manu_name	product.vendor_name
Customer Dimension Table:		
customer	stores7:orders.customer_num	customer.customer_code
customer name	stores7:customer.fname plus stores7:customer.lname	customer.customer_name
company	stores7:customer.company	customer.company_name

(1 of 2)

Business term	Data source	Table.Column name
Geography Dimension Table:		
district code	generated	geography.district_code
district	stores7:customer.city	geography.district_name
state	stores7:customer.state	geography.state_code
state name	stores7.state.sname	geography.state_name
region	derived: If state = "CA" THEN region = 1, ELSE region = 2	geography.region
Time Dimension Table:		
time code	generated	time.time_code
order date	stores7:orders.order_date	time.order_date
month	derived from order date generated	time.month_name time.month.code
quarter	derived from order date generated	time.quarter_name time.quarter_code
year	derived from order date	time.year

(2 of 2)

Several files with a **.unl** suffix contain the data that is loaded into the **sales_demo** database. The files that contain the SQL statements that create and load the database have a **.sql** suffix.

UNIX

When your database server runs on the UNIX operating system you can access the ***.sql** and ***.unl** files from the directory **$INFORMIXDIR/demo/dbaccess**. ♦

WIN NT

When your database server runs on the Windows NT operating system you can access the ***.sql and *.unl** files from the directory **%INFORMIXDIR%\demo\dbaccess**. ♦

You can also access the files that have a **.sql** suffix as command files from DB-Access.

Loading Data into the Dimensional Database

An important step when you implement a dimensional database is to develop and document a load strategy. This section shows the LOAD and INSERT statements that you can use to populate the tables of the **sales_demo** database.

Tip: *In a live data-warehousing environment, you typically do not use the LOAD or INSERT statements to load large amounts of data to and from Informix databases. Dynamic Server and Dynamic Server with AD and XP Options provide different features for high-performance loading and unloading of data.*

IDS

When you create a database with Dynamic Server, you can use the High-Performance Loader (HPL) to perform high-performance loading and unloading. ♦

AD/XP

When you create a database with Dynamic Server with AD and XP Options, you can use *external tables* to perform high-performance loading and unloading. ♦

For information about high-performance loading, see your *Administrator's Guide* or high-performance loader documentation.

The following statement loads the **time** table with data first so that you can use it to determine the time code for each row that is loaded into the **sales** table.

```
LOAD FROM 'time.unl'
INSERT INTO time;
```

The following statement loads the **geography** table. Once you load the **geography** table, you can use the district code data to load the **sales** table.

```
INSERT INTO geography(district_name, state_code,
    state_name)
SELECT DISTINCT c.city, s.code, s.sname
    FROM stores7:customer c, stores7:state s
    WHERE c.state = s.code;
```

The following statements add the region code to the **geography** table.

```
UPDATE geography
    SET region = 1
    WHERE state_code = 'CA';
UPDATE geography
    SET region = 2
    WHERE state_code <> 'CA';
```

The following statement loads the **customer** table.

```
INSERT INTO customer
    (customer_code, customer_name, company_name)
SELECT c.customer_num, trim(c.fname) ||' '|| c.lname,
    c.company
FROM stores7:customer c;
```

The following statement loads the **product** table.

```
INSERT INTO product
    (product_code, product_name, vendor_code,
    vendor_name,product_line_code, product_line_name)
SELECT a.catalog_num,
    trim(m.manu_name)||' '||s.description,
    m.manu_code, m.manu_name,
    s.stock_num, s.description
FROM stores7:catalog a, stores7:manufact m,
    stores7:stock s
    WHERE a.stock_num = s.stock_num
    AND a.manu_code = s.manu_code
    AND s.manu_code = m.manu_code;
```

The following statement loads the **sales** fact table with one row for each product per customer per day per district. The cost from the **cost** table is used to calculate the total cost (cost * quantity).

```
INSERT INTO sales
    (customer_code, district_code, time_code,
    product_code, units_sold, cost, revenue,
    net_profit)
SELECT
    c.customer_num, g.district_code, t.time_code,
    p.product_code, SUM(i.quantity),
    SUM(i.quantity * x.cost), SUM(i.total_price),
    SUM(i.total_price) - SUM(i.quantity * x.cost)
FROM stores7:customer c, geography g, time t,
    product p,stores7:items i, stores7:orders o, cost x
WHERE c.customer_num = o.customer_num
AND o.order_num = i.order_num
AND p.product_line_code = i.stock_num
AND p.vendor_code = i.manu_code
AND t.order_date = o.order_date
AND p.product_code = x.product_code
AND c.city = g.district_name
GROUP BY 1,2,3,4;
```

For your convenience, the file called **loaddw.sql** contains all the preceding INSERT and LOAD statements, and is located in the directory **$INFORMIXDIR/demo/dbaccess**.

Using Command Files to Create the sales_demo Database

You can use command files to create and populate the tables of the **sales_demo** database. You execute the **createdw.sql** and **loaddw.sql** files to implement the **sales_demo** database.

The **sales_demo** dimensional database uses data from the **stores7** database, so you must create both databases to implement the **sales_demo** database.

To create and populate the sales_demo database

1. Set the **INFORMIXDIR** environment variable so that it contains the name of the directory in which your Informix products are installed.
2. Set the **INFORMIXSERVER** environment variable to the name of the default database server.
3. Enter the following command to make the directory in which your Informix products are installed the current directory:

   ```
   cd $INFORMIXDIR
   ```
4. Enter the following command to create the **stores7** database and copy the demonstration database examples into your current directory:

   ```
   dbaccessdemo7
   ```
5. Execute the following command to create the **sales_demo** database:

   ```
   dbaccess stores7 createdw.sql
   ```
6. Execute the following command file to load the **sales_demo** database:

   ```
   dbaccess sales_demo loaddw.sql
   ```

   The **stores7** database must already exist to load the **sales_demo** database because the **loaddw.sql** file adds rows to the **items** and **orders** tables of **stores7**. The **items** and **orders** tables are then used to load certain tables in the **sales_demo** database.

Important: *Execution of the **loaddw.sql** file changes the data in the **stores7** database. Consequently, the query results that you get with this modified **stores7** database might differ from the SQL examples based on **stores7** data and documented in Informix manuals.*

Testing the Dimensional Database

You can create SQL queries to retrieve the data necessary for the standard reports listed in the business-process summary (see the "Summary of a Business Process" on page 6-16). Use the following ad hoc queries to test that the dimensional database was properly implemented.

The following statement returns the monthly revenue, cost, and net profit by product line for each vendor:

```
SELECT vendor_name, product_line_name, month_name,
    SUM(revenue) total_revenue, SUM(cost) total_cost,
    SUM(net_profit) total_profit
FROM product, time, sales
WHERE product.product_code = sales.product_code
    AND time.time_code = sales.time_code
GROUP BY vendor_name, product_line_name, month_name
ORDER BY vendor_name, product_line_name;
```

The following statement returns the revenue and units sold by product, by region, and by month:

```
SELECT product_name, region, month_name,
    SUM(revenue), SUM(units_sold)
FROM product, geography, time, sales
WHERE product.product_code = sales.product_code
    AND geography.district_code = sales.district_code
    AND time.time_code = sales.time_code
GROUP BY product_name, region, month_name
ORDER BY product_name, region;
```

The following statement returns the monthly customer revenue:

```
SELECT customer_name, company_name, month_name,
    SUM(revenue)
FROM customer, time, sales
WHERE customer.customer_code = sales.customer_code
    AND time.time_code = sales.time_code
GROUP BY customer_name, company_name, month_name
ORDER BY customer_name;
```

The following statement returns the quarterly revenue per vendor:

```
SELECT vendor_name, year, quarter_name, SUM(revenue)
    FROM product, time, sales
    WHERE product.product_code = sales.product_code
    AND time.time_code = sales.time_code
    GROUP BY vendor_name, year, quarter_name
    ORDER BY vendor_name, year;
```

AD/XP

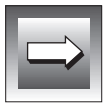
Logging and Nonlogging Tables for Dynamic Server with AD and XP Options

This section describes the different Dynamic Server with AD and XP Options table types that can be particularly useful in data-warehousing environments. Dynamic Server with AD and XP Options logs tables by default, the same way that Dynamic Server logs tables. However, data-warehousing environments and other applications that involve large amounts of data (and few or no inserts, updates, or deletes) often require a combination of logged and nonlogged tables in the same database. In many cases, temporary tables are insufficient because they do not persist after the database session ends. To meet the need for both logging and nonlogging tables, Dynamic Server with AD and XP Options supports the following types of permanent tables and temporary tables:

- Raw permanent tables (nonlogging)
- Static permanent tables (nonlogging)
- Operational permanent tables (logging)
- Standard permanent tables (logging)
- Scratch temporary tables (nonlogging)
- Temp temporary tables (logging)

If you issue the CREATE TABLE statement and you do not specify the table type, you create a standard permanent table.

For information about the syntax you use to create these tables, see the CREATE TABLE statement in the *Informix Guide to SQL: Syntax*. For the syntax you use to change between table types, see the ALTER TABLE statement in the *Informix Guide to SQL: Syntax*.



Important: *A coserver can use and access only its own dbspaces for temporary space. Although temporary tables can be fragmented explicitly across dbspaces like permanent tables, a coserver inserts data only into the fragments that it manages.*

Choosing Table Types

The individual tables in a data-warehousing environment often have different requirements. You can answer the following questions to help determine the appropriate table type to use for your tables:

- Does the table require indexes?
- What constraints does the table need to define?
- What is the refresh and update cycle on the table?
- Is the table a read-only table?
- Does the table need to be logged?

Figure 7-2 lists the properties of the six types of tables that Dynamic Server with AD and XP Options supports and shows how you can use external tables to load these types of tables. Use this information to select a table type to match the specific requirements of your tables.

Figure 7-2
Characteristics of the Table Types for Dynamic Server with AD and XP Options

Type	Permanent	Logged	Indexes	Light Append Used	Rollback Available	Recoverable	Restorable from Archive	External Tables Load Mode
SCRATCH	No	No	No	Yes	No	No	No	Express or deluxe load mode
TEMP	No	Yes	Yes	Yes	Yes	No	No	Express or deluxe load mode
RAW	Yes	No	No	Yes	No	No	No	Express or deluxe load mode

(1 of 2)

Type	Permanent	Logged	Indexes	Light Append Used	Rollback Available	Recoverable	Restorable from Archive	External Tables Load Mode
STATIC	Yes	No	Yes	No	No	No	No	None
OPERATIONAL	Yes	Yes	Yes	Yes	Yes	Yes	No	Express or deluxe load mode
STANDARD	Yes	Yes	Yes	No	Yes	Yes	Yes	Deluxe load mode

(2 of 2)

Scratch and Temp Tables

Scratch tables are nonlogging temporary tables that do not support indexes, constraints, or rollback.

Temp tables are logged tables, although they also support bulk operations such as light appends. (Express mode loads use *light appends*, which bypass the buffer cache. Light appends eliminate the overhead associated with buffer management but do not log the data.) Temp tables support indexes, constraints, and rollback.

Tip: *SELECT...INTO TEMP and SELECT...INTO SCRATCH statements are parallel across coservers, just like ordinary inserts. Dynamic Server with AD and XP Options automatically supports fragmented temporary tables across nodes when those tables are explicitly created with SELECT...INTO TEMP and SELECT...INTO SCRATCH.*

Dynamic Server with AD and XP Options creates explicit temporary tables according to the following criteria:

- If the query that you use to populate the TEMP table produces no rows, the database server creates an empty, unfragmented table.
- If the rows that the query produces do not exceed 8 kilobytes, the temporary table resides in only one dbspace.
- If the rows exceed 8 kilobytes, Dynamic Server with AD and XP Options creates multiple fragments and uses a round-robin fragmentation scheme to populate them.

Raw Permanent Tables

Raw tables are nonlogging permanent tables that use light appends. Express-mode loads use *light appends*, which bypass the buffer cache. You can load a raw table with express mode. For information about express-mode loads, see your *Administrator's Guide*.

Raw tables support updates, inserts, and deletes but do not log them. Raw tables do not support index or referential constraints, rollback, recoverability, or restoration from archives.

Use raw tables for the initial data loading and scrubbing. Once these steps are completed, alter the table to a higher level. For example, if an error or failure occurs while you are loading a raw table, the resulting data is whatever was on the disk at the time of the failure.

In a data-warehousing environment, you might choose to create a fact table as a raw table when both of the following conditions are true:

- The fact table does not need to specify constraints and indexes, which are enforced by some different mechanisms.
- Creating and loading the fact table is not a costly job. The fact tables could be useful but not critical for decision support, and if data is lost you can easily reload the table.

Static Permanent Tables

Static tables are nonlogging, read-only permanent tables that do not support insert, update, and delete operations. When you anticipate no insert, update, or delete operations on the table, you might choose to create the table as a static table. With a static table, you can create and drop nonclustered indexes and referential constraints because they do not affect the data.

Static tables do not support rollback, recoverability, or restoration from archives. Their advantage is that the server can use light scans and avoid locking when you execute queries because static tables are read-only.

Tip: *Static tables are important when you want to create a table that uses GK indexes because a static table is the only table type that supports GK indexes.*

Operational Permanent Tables

Operational tables are logging permanent tables that use light appends and do not perform record-by-record logging. They allow fast update operations.

You can roll back operations or recover after a failure with operational tables, but you cannot restore them reliably from an archive of the log because the bulk insert records that are loaded are not logged. Use operational tables in situations where you derive data from another source so restorability is not an issue, but where you do not require rollback and recoverability.

You might create a fact table as an operational table because the data is periodically refreshed. Operational tables support express load mode (in the absence of indexes and constraints) and data is recoverable.

Standard Permanent Tables

A standard table is the same as a table in a logged database that you create with Dynamic Server. All operations are logged, record by record, so you can restore standard tables from an archive. Standard tables support recoverability and rollback.

If the update and refresh cycle for the table is infrequent, you might choose to create a standard table type, as you need not drop constraints or indexes during a refresh cycle. Building indexes is time consuming and costly, but necessary.

Tip: *Standard tables do not use light append, so you cannot use express-load mode when you use external tables to perform the load.*

Switching Between Table Types

Use the ALTER TABLE command to switch between types of permanent tables. If the table does not meet the restrictions of the new type, the alter fails and produces an explanatory error message. The following restrictions apply to table alteration:

- You must drop indexes and referential constraints before you alter a table to a RAW type.
- You must perform a level-0 archive before you alter a table to a STANDARD type, so that the table meets the full recoverability restriction.
- You cannot alter a temp or scratch temporary table.

AD/XP

Indexes for Data-Warehousing Environments

In addition to conventional (B-tree) indexes, Dynamic Server with AD and XP Options provides the following indexes that you can use to improve ad hoc query performance in data-warehousing environments:

- Bitmap indexes

  A bitmap index uses less disk space than a conventional B-tree index. With a bitmap index, storage efficiency increases as the distance between rows that contain the same key decreases.

  You can use a bitmap index when both of the following conditions are true:

  - The key values in the index contain many duplicates.
  - More than one column in the table has an index that the optimizer can use to improve performance on a table scan.

- Generalized-key (GK) indexes

  GK indexes allow you to store the result of an expression, selection of a data set, or intersect of data sets from joined tables as a key in a B-tree or bitmap index, which can be useful in specific queries on one or more large tables.

  To create a GK index, all tables involved should be static tables.

To improve indexing efficiency, Dynamic Server with AD and XP Options also supports the following functionality:

- Automatically combine indexes for use in the same table access

  You can combine multi-column indexes with single-column indexes.

- Read a table with an access method known as a Skip Scan

  When it scans rows from a table, the database server only reads rows that the index indicates, and reads rows in the order that they appear in the database. The *skip scan* access method guarantees that no page is read twice. Pages are read sequentially, not randomly, which reduces I/O resource requirements. The skip scan also reduces CPU requirements because filtering on the index columns is unnecessary.

- Use a hash semi-join to reduce the work to process certain multitable joins

  A hash semi-join is especially useful with joins that typify queries against a star schema where one large (fact) table is joined with many small (dimension) tables. The hash-join can effectively reduce the set of rows as much as possible before the joins begin.

An analysis of the types of queries you anticipate running against your database can help you decide the type of indexes to create. For information about indexes and indexing methods you can use to improve query performance, see your *Performance Guide*.

Using GK Indexes in a Data-Warehousing Environment

You can create GK indexes when you anticipate frequent use of a particular type of query on a table. The following examples illustrate how you can create and use GK indexes for queries on one or more large tables. The examples are based on tables of the **sales_demo** database.

Defining a GK Index on a Selection

Suppose a typical query on the **sales** fact table returns values where state = "CA". To improve the performance for this type of query, you can create a GK index that allows you to store the result of a select statement as a key in an index. The following statement creates the **state_idx** index, which can improve performance on queries that restrict a search by geographic data.

```
CREATE GK INDEX state_idx on geography
    (SELECT district_code FROM geography
        WHERE state_code = "CA")
```

The database server can use the **state_idx** index on the following type of query that returns revenue and units sold by product, by region, and by month where state = "CA". The database server uses the **state_idx** index to retreive rows from the **geography** table where state = "CA" to improve query performance overall.

```
SELECT product_name, region, month_name, SUM(revenue),
SUM(units_sold)
FROM product, geography, time, sales
    WHERE product.product_code = sales.product_code AND
    geography.district_code = sales.district_code AND
    state_code = "CA" AND time.time_code = sales.time_code
    GROUP BY product_name, region, month_name
    ORDER BY product_name, region;
```

Defining a GK Index on an Expression

You can create a GK index that allows you to store the result of an expression as a key in an index. The following statement creates the **cost_idx** index, which can improve performance for queries against the **sales** table that include the cost of the products sold.

```
CREATE GK INDEX cost_idx on sales
    (SELECT units_sold * cost FROM sales);
```

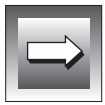
The database server can use the **cost_idx** index for the following type of query that returns the names of customers who have spent more than $10,000.00 on products.

```
SELECT customer_name
FROM sales, customer
WHERE sales.customer_code = customer.customer_code
    AND units_sold * cost > 10000.00;
```

Defining a GK Index on Joined Tables

You can create a GK index that allows you to store the result of an intersect of data sets from joined tables as a key in an index. Suppose you wish to create a GK index on year data from the **time** dimension table for each entry in the **sales** table. The following statement creates the **time_idx** index:

```
CREATE GK INDEX time_idx on sales
(SELECT year FROM sales, time
    WHERE sales.time_code = time.time_code )
```



Important: *To create the preceding GK index, the* ***time_code*** *column of the* ***sales*** *table must be a foreign key that references the* ***time_code*** *column (a primary key) in the* ***time*** *table.*

The database server can use the **time_idx** index on the following type of query that returns the names of customers who purchased products after 1996.

```
SELECT customer_name
FROM   sales, customer, time
WHERE  sales.time_code = time.time_code AND year > 1996
        AND sale.customer_code = customer.customer_code
```

Managing Databases

Section III

Chapter

8

Granting and Limiting Access to Your Database

Controlling Access to Databases 8-4
Securing Confidential Data 8-4

Granting Privileges 8-5
Database-Level Privileges 8-5
Connect Privilege 8-5
Resource Privilege 8-6
Database-Administrator Privilege 8-7
Ownership Rights 8-7
Table-Level Privilege 8-8
Access Privileges 8-8
Index, Alter, and References Privileges 8-10
Column-Level Privileges 8-10
Procedure-Level Privileges 8-12
Automating Privileges 8-13
Automating with a Command Script 8-14
Using Roles with Dynamic Server 8-14

Using Stored Procedures to Control Access to Data 8-17
Restricting Data Reads 8-18
Restricting Changes to Data 8-19
Monitoring Changes to Data 8-20
Restricting Object Creation 8-21

Using Views 8-22
Creating Views 8-23
Duplicate Rows from Views 8-24
Restrictions on Views 8-25
When the Basis Changes 8-25

Modifying with a View 8-26
Deleting with a View 8-27
Updating a View 8-27
Inserting into a View. 8-28
Using the WITH CHECK OPTION Keywords 8-29

Privileges and Views 8-30
Privileges When Creating a View. 8-30
Privileges When Using a View. 8-31

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 of the data. You can restrict access to data at the following levels, which are the subject of this chapter:

- 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 statement to write and compile a stored procedure, 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.

Controlling Access to Databases

The normal database-privilege mechanisms are based on the GRANT and REVOKE statements. They are discussed in "Granting Privileges" on page 8-5. However, you can sometimes use the facilities of the operating system as an additional way to control access to a database.

Securing Confidential Data

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 on-line. 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 files as well as the table files.

Granting Privileges

The authorization to use a database is called a *privilege*. For example, the authorization to use a database is called the Connect privilege, and the authorization to insert a row into a table is called the Insert privilege. You 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. You use the REVOKE statement to revoke privileges on a database, table, view, or procedure or to revoke a role from a user or another role. A *role* is a classification or work task that the DBA assigns, such as **payroll**. Assignment of roles makes management of privileges convenient.

AD/XP

Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options does not support roles. ♦

Two groups of privileges control the actions a user can perform on data. These include database-level privileges, which affect the entire database, and table-level privileges, which relate to individual tables. In addition to these two groups, procedure-level privileges determine who can execute a procedure.

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

Database-Level Privileges

The three levels of database privilege provide an overall means of controlling who accesses a database.

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 a stored procedure, 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.

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.

Prior to executing a statement, the database server determines whether a user has the necessary privileges. The information is in the system catalog; see "Privileges in the System Catalog" on page 8-9.

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 stored procedures, thus permanently allocating disk space.

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
- Alter the NEXT SIZE (but no other attribute) of the system catalog tables, and insert, delete, or update rows of any system catalog table except **systables**

Warning: *Although users with the DBA privilege can modify most system catalog tables, Informix strongly recommends that you do not update, delete, or insert any rows in them. Modifying the system catalog tables can destroy the integrity of the database. You cannot use the ALTER TABLE statement to modify the size of the next extent of system catalog tables.*

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 an object has all rights to that object and can alter or drop it without additional privileges.

AD/XP

For GK indexes, ownership rights are handled somewhat differently than they are for other objects. Any table that appears in the FROM clause of a GK index cannot be dropped until that GK index is dropped, even when someone other than the creator of the table creates the GK index. ♦

Table-Level Privilege

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 access to the contents of the table. The Index privilege controls index creation. The Alter privilege controls the authorization to change the table definition. The References privilege controls the 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. When you 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

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 privileges are recorded in the **sysusers** table, in which the primary key is user ID, and the only other column contains a single character `C`, `R`, or `D` for the privilege level. A grant to the keyword of PUBLIC is reflected as a user name of **public** (lowercase).

Table-level privileges are recorded in **systabauth**, which uses a composite primary key of the table number, grantor, and grantee. In the **tabauth** column, the privileges are encoded in the list that Figure 8-1 shows.

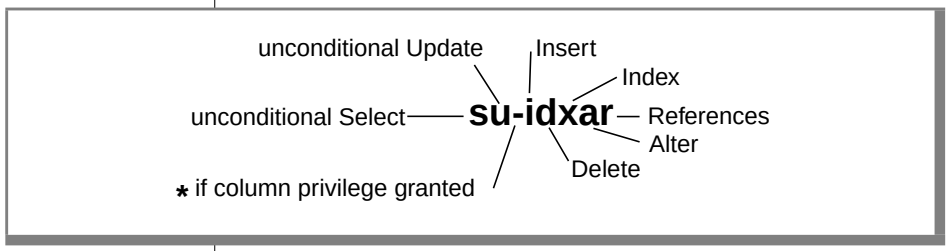


Figure 8-1
List of Encoded Privileges

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.

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, you 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 discusses a way to limit the view of managers to their employees only.) For the first-line managers who carry out performance reviews, you could 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 whoever is trusted to alter salary levels:

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

For the clerks in the Human Resources department, you could 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'
```

Procedure-Level Privileges

You can apply the Execute privilege on a procedure to authorize nonowners to run a procedure. If you create a procedure in a database that is not ANSI compliant, the default procedure-level privilege is PUBLIC; you do not need to grant the Execute privilege to specific users unless you have first revoked it. If you create a procedure 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 stored procedure that is named **read_address**:

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

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

For more information on stored procedure privileges, see the *Informix Guide to SQL: Tutorial*.

Automating 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 need, 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 needs the following privileges:

- The Select and limited Update privilege 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 the 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 procedures for each class, one to grant the class to a user and one to revoke it.

Automating 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 and Relational Object Manager 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 DB-Access or Relational Object Manager that you are using. You want to 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 DB-Access or Relational Object Manager 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.

IDS

Using Roles with Dynamic Server

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 privileges of many users by treating them as members of a class.

For example, you can create a role called *news_mes* that grants connect, insert, and delete privileges for the databases that handle company news and messages. When a new employee arrives, you need only add that person to the role *news_mes*. The new employee acquires the privileges of the role *news_mes*. This process also works in reverse. To change the privileges of all the members of *news_mes*, change the privileges of the role.

Creating a Role

To start the role creation process, determine the name of the role along with the connections and privileges you want to grant. Although the connections and privileges are strictly in your domain, you need to consider some factors when you name a role. Do not use any of the following words as role names.

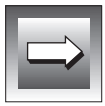
alter	default	index	null	resource
connect	delete	insert	public	select
DBA	execute	none	references	update

A role name must be different from existing role names in the 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 machine. To make sure your role name is unique, check the names of the users in the shared memory structure who are currently using the database as well as the following system catalog tables:

- **sysusers**
- **systabauth**
- **syscolauth**
- **sysprocauth**
- **sysfragauth**
- **sysroleauth**

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.



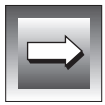
Important: *The scope of a role is the current database only, so when you execute a SET ROLE statement, the role is set in the current database only.*

Manipulating User Privileges and Granting 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 use the WITH GRANT OPTION clause only when you are granting privileges to a user.

For example, the following query returns an error because you are granting privileges to a role with the grantable option:

```
GRANT SELECT on tab1 to rol1
    WITH GRANT OPTION
```



Important: *Do not use the WITH GRANT OPTION clause of the GRANT statement when you grant privileges to a role. Only a user can grant privileges to other users.*

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

Users Need to Enable Roles

After the DBA grants privileges and adds users to a role, you must use the SET ROLE statement in a database session to enable the role. Unless you enable the role, you are limited to the privileges that are associated with **public** or the privileges that are directly granted to you because you own the object.

Confirming Membership In Roles and Dropping 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 not available. Issue queries against the **sysroleauth** and **sysusers** tables to find who is authorized for which table and how many roles exist.

After you determine which users are members of 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** catalog table as a role can be destroyed.
- You must have DBA privileges, or you must be given the grantable option in the role to drop a role.

Using Stored Procedures to Control Access to Data

You can use a stored procedure to control access to individual tables and columns in the database. Use a procedure to accomplish various degrees of access control. (Stored procedures are fully described in the *Informix Guide to SQL: Tutorial*.) A powerful feature of stored procedure language (SPL) is the ability to designate a stored procedure as a DBA-privileged procedure. When you write a DBA-privileged procedure, you can allow users who have few or no table privileges to have DBA privileges when they execute the procedure. In the procedure, users can carry out specific tasks with their temporary DBA privilege. The DBA-privileged feature 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 a stored procedure so that you have complete control over how tables, indexes, and views are built.

Restricting Data Reads

The procedure 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 procedure 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 stored procedure (or set of stored procedures).
- Your stored procedure (or set of stored procedures) 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;
```

Restricting Changes to Data

When you use stored procedures, you can restrict changes made to a table. Channel all changes through a stored procedure. The stored procedure 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 stored procedure.
- Your stored procedure performs the deletion.

Write a stored 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;
```

Monitoring Changes to Data

When you use stored procedures, 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 of each time a change is made.

You can monitor all the changes a single user makes to the database. Channel all changes through stored procedures 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 a stored procedure.
- Your stored procedure performs the deletion and records that a certain user makes a change.

Write a stored procedure similar to the following example, 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**.

UNIX

```
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 procedure 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;
```

♦

Restricting Object Creation

To put restraints on what objects are built and how they are built, use stored procedures 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 a stored procedure (or set of stored procedures).
- Your stored procedure (or set of stored procedures) creates tables, indexes, and views in the way you defined them. You might use such a procedure to set up a training database environment.

Your procedure 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 that user. For the **all_objects** procedure, whoever executes the procedure owns the two tables and the index.

Using 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. The database server merges the SELECT statement of the user with the one defined for the view and then actually performs the combined statements.

The result has the appearance of a table; it is similar enough to a table that a view even can be based on other views, or on joins of tables and other views.

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 (discussed on page 8-29) to enforce constraints.

- To provide access to derived data without having to store 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 need to repeat them.

Creating Views

The following example creates a view based on a table in the demonstration 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.

GLS

The following example creates a view based on a table that is available when a locale other than the default U.S. English locale using the ISO8859-1 code set is enabled. In the example, the view, column, and table names contain non-English characters.

```
CREATE VIEW çà_va AS
SELECT numéro, nom FROM abonnés;
```

♦

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

```
CREATE VIEW full_addr AS
SELECT address1, address2, city, state.sname, zipcode
    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
    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 a discussion about how to modify with views, see page 8-26.

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.

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 select 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. Primary and candidate keys are discussed in Chapter 2, "Building a Relational Data Model.")

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 could 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 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 it 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 it 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 proper column. However, the names of columns in the views themselves are not changed. For an example of this, 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 last names 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 new column of the same name. If you do this, views based on that table continue to work. They retain their original sequence of columns.

The database server permits you to 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.

Modifying 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.

No modification is possible on a view when its SELECT statement contains any of the following features:

- A join of two or more tables

  Many anomalies arise if the database server tries to distribute modified data correctly across the joined tables.

- An aggregate function or the GROUP BY clause

  The rows of the view represent many combined rows of data; the database server cannot distribute modified data into them.

- The DISTINCT keyword or its synonym UNIQUE

  The rows of the view represent a selection from among possibly many duplicate rows; the database server cannot tell which of the original rows should receive the modification.

- The UNION keyword

  The rows of the view do not carry a tag that identifies the table that produces the row. Therefore, for an update or delete operation, the database server is unable to identify the table in which the row should be updated or deleted. In the case of an insert operation, the database server is unable to identify the table in which to insert the row.

When a view avoids all these restricted features, each row of the view corresponds to exactly one row of one table. Such a view is *modifiable*. (Of course, particular users can modify a view only if they have suitable privileges. Privileges on views are discussed beginning on page 8-30.)

Deleting with a View

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

Updating a View

You can use a modifiable view with an UPDATE statement as if it were a table. However, a modifiable view can still contain derived columns; that is, columns that are produced by expressions in the select list of the CREATE VIEW statement. You cannot update derived columns (sometimes called *virtual* columns).

When you derive a column from a simple arithmetic combination of a column with a constant value (for example, `order_date + 30`), the database server can, in principle, figure out how to invert the expression (in this case, by subtracting 30 from the update value) and perform the update. However, much more complicated expressions are possible, most of which cannot easily be inverted. Therefore, the database server does not support updating any derived column.

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

```
CREATE VIEW call_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 call_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 as long as 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.

Inserting into a View

You can insert rows into a view, provided that 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, and the database server cannot tell how to distribute an inserted value through an expression. An attempt to insert into the **call_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. However, the database server uses null as the value for any column that is not exposed by the view. If such a column does not allow nulls, an error occurs, and the insert fails.

Using 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, you can add the clause WITH CHECK OPTION 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.

Important: *You cannot include the WITH CHECK OPTION clause when a UNION operator is included in the view definition.*

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

```
CREATE VIEW call_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 call_response SET user_id = 'lenora'
      WHERE received BETWEEN TODAY AND TODAY-7
```

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

```
CREATE VIEW call_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 by **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.

Privileges and Views

When you *create* a view, the database server tests your privileges on the underlying tables and views. When you *use* a view, only your privileges with regard to the view are tested.

Privileges When Creating a View

The database server tests to make sure that you have all the privileges that you need 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.

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 public access 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 8-10 shows how to grant privileges directly on the **hr_data** table. The following examples take a different approach. Assume that when the table was created, the following 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 need access to all columns and they need 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
```

Index

A

Access privileges 8-8
Accessing data in a fragmented table 5-42
ADD ROWID clause, of ALTER TABLE 5-43
Adding a table fragment 5-34
Aggregate function, restrictions in modifiable view 8-26
ALTER FRAGMENT statement
  adding rowid column 5-43
  INIT clause, using 5-31
  with ADD clause 5-34
  with ATTACH clause 5-35
  with DETACH clause 5-36
  with DROP clause 5-35
  with INIT clause 5-31
  with MODIFY clause 5-32
Alter privilege 8-10
ALTER TABLE statement
  ADD ROWIDS clause 5-43
  changing column data type 3-23
  DROP ROWIDS clause 5-44
  privilege for 8-10
  switching between table types 7-18
ANSI compliance
  described 1-4
  determining 1-5
  icon Intro-11
  level Intro-16
ANSI-compliant database
  buffered logging restricted in 4-6
  designating 1-5
  effect on
    cursor behavior 1-9
    decimal data type 1-8
    default isolation level 1-8
    escape characters 1-9
    object privileges 1-7
    owner naming 1-7
    SQLCODE 1-9
    transaction logging 1-7
    transactions 1-6
  owner naming 1-7
  privileges 1-7
  reason for creating 1-4
  SQL statements allowed 1-10
  table privileges 8-8
Archive, and fragmentation 5-7
Asian Language Support, compatibility with version 7.2 products 1-10
Attribute
  identifying 2-17
  important qualities of 2-17
  nondecomposable 2-17
Availability, improving with fragmentation 5-7

B

Backup-and-restore operations, fragmentation with 5-7
Bitmap index, description of 7-18
Buffered logging 4-6
Building a relational data model 2-3 to 2-21

BYTE data type
  description of 3-21
  using 3-22

C

Candidate key 2-26
Cardinality
  constraint 2-11
  in relationship 2-15
Chaining synonyms 4-10
CHAR data type 3-17
CHARACTER VARYING data type 3-18
Codd, E. F. 2-35
Code set 1-10
Column
  defining 2-23
  of a fragmented table, modifying 5-30
Column-level privileges 8-10
Command script, creating a database 4-11
Comment icons Intro-9
Compliance
  icons Intro-11
  with industry standards Intro-16
Composite key 2-26
Concurrency, improving with fragmentation 5-6
Connect privilege 8-5
Connectivity in relationship 2-10, 2-13, 2-20
Constraint
  cardinality 2-11
  defining domains 3-3
CREATE DATABASE statement
  implementing
    dimensional data model 7-3
    relational data model 4-4
  in command script 4-11
CREATE TABLE statement
  description of 4-6
  in command script 4-11
  SCRATCH 5-37
  setting initial SERIAL value 3-8
  TEMP 5-37
  with FRAGMENT BY EXPRESSION clause 5-11
  WITH ROWIDS clause 5-43
CREATE VIEW statement
  restrictions on 8-25
  using 8-23
  WITH CHECK OPTION keywords 8-29

D

Data
  loading with dbload utility 4-12
  loading with external tables 4-12
Data mart, description of 6-4
Data model
  attribute 2-17
  building
    dimensional 6-15 to 6-27
    relational 2-3 to 2-21
  defining relationships 2-9
  description of 2-3
  dimensional 6-9
  entity relationship 2-5
  many-to-many relationship 2-13
  one-to-many relationship 2-13
  one-to-one relationship 2-13
  relational 2-3
  telephone-directory example 2-7
Data type
  BYTE 3-21
  changing with ALTER TABLE statement 3-23
  CHAR 3-17
  CHARACTER VARYING 3-18
  chronological 3-12
  DATE 3-13
  DATETIME 3-14
  DECIMAL 3-10, 3-11
  fixed-point 3-11
  floating-point 3-9
  INTEGER 3-7
  INTERVAL 3-15
  MONEY 3-11
  NCHAR 3-17
  numeric 3-7
  NVARCHAR 3-18
  REAL 3-9
  SERIAL 3-7
  SMALLFLOAT 3-9
  TEXT 3-21
  VARCHAR 3-18
Data warehouse, description of 6-4
Database
  external database, allowing views on 8-26
  naming 4-4
  populating new tables 4-12
Database administrator (DBA) 8-7
Database server, allowing views on external databases 8-26
Database-level privileges
  Connect privilege 8-5
  database-administrator privilege 8-7
  description of 8-5
  Resource privilege 8-6
Data-warehousing model. *See* Demonstration database, sales_demo.
Data, accessing in fragmented tables 5-42
DATE data type
  customizing format of 3-13
  description of 3-13
  display format 3-13
DATETIME data type
  description of 3-14
  display format 3-16
  international date and time formats 3-16
DB-Access
  creating database with 4-11
  UNLOAD statement 4-12
DBA. *See* Database administrator.
DBDATE environment variable 3-13
dbload utility, loading data into a table 4-12
DBMONEY environment variable 3-12
dbschema utility 4-11
dbslice, role in fragmentation 5-6
dbspace
  role in fragmentation 5-3
  selecting with CREATE DATABASE statement 4-5

DBSPACETEMP configuration parameter 5-38
DBSPACETEMP environment variable 5-38
DBTIME environment variable 3-16
DECIMAL data type
  fixed-point 3-11
  floating-point 3-10
Default locale Intro-5
Default value
  defined 3-23
  specifying for a column 3-24
Delete privilege 8-8, 8-31
DELETE statement
  applied to view 8-27
  privilege for 8-5, 8-8
Demonstration database Intro-5
Derived data, produced by view 8-23
Descriptor column 2-25
Determining ANSI compliance 1-5
Dimension table
  choosing attributes for 6-26
  description of 6-14
Dimensional data model
  building 6-15 to 6-27
  dimension elements 6-12
  dimension tables 6-14
  dimensions 6-11
  fact table 6-10
  implementing 7-3
  measures, definition of 6-10
Dimensional database, sales_demo 7-4
DISTINCT keyword, restrictions in modifiable view 8-27
Distribution scheme
  changing the number of fragments 5-31
  description of 5-4
  expression-based
    fragment elimination in 5-22
    using 5-11
    with arbitrary rule 5-12
    with range rule 5-12
  hybrid
    description of 5-4
    fragment elimination in 5-24
    using 5-14
  round-robin
    description of 5-4
    using 5-11
  system-defined hash
    description of 5-4
    fragment elimination in 5-23
    using 5-13
  types
    for Dynamic Server 5-9
    for Dynamic Server with AD and XP Options 5-10
Documentation conventions
  command-line Intro-11
  icon Intro-9
  sample-code Intro-11
  typographical Intro-8
Documentation notes
  location Intro-14
  program item Intro-15
Documentation, types of
  documentation notes Intro-14
  error message files Intro-13
  machine notes Intro-14
  on-line manuals Intro-12
  printed manuals Intro-13
  related reading Intro-15
  release notes Intro-14
Domain
  characteristics 2-24, 3-3
  constraints 3-3
  defined 2-24
  of column 3-3
DROP ROWIDS clause, of ALTER TABLE 5-44
Dropping a table fragment 5-35
Dynamic Server with AD and XP Options 3-21

E

Entity
  attributes associated with 2-17
  business rules 2-5
  criteria for choosing 2-8
  defined 2-5
  important qualities of 2-6
  in telephone-directory example 2-9
  naming 2-5
  represented by a table 2-25
Entity occurrence, defined 2-18
Entity-relationship diagram
  connectivity 2-20
  discussed 2-19
  meaning of symbols 2-19
  reading 2-20
Environment, Non-U.S. English 1-10
en_us.8859-1 locale Intro-5
Error message files Intro-13
Even distribution 5-11
Existence dependency 2-10
EXISTS keyword, use in condition subquery 8-30
Expression fragment, how searches are done 5-15
Expression-based distribution scheme
  and fragment elimination 5-16
  arbitrary rule 5-12
  description of 5-4
  using 5-11
  with range rule 5-12
Extension, to SQL
  symbol for Intro-11
  with ANSI-compliant database 1-10
External tables, loading data with 4-12, 5-7, 7-9

F

Fact table
  description of 6-10
  determining granularity of 6-18
  granularity of 6-11
Feature icons Intro-10
Features, product Intro-6
finderr utility Intro-13
First normal form 2-32
Fixed point 3-11
Flex operator 5-38

Flex temporary table 5-38
  definition of 5-38
  fragmentation of 5-38
Flex temporary table operator 5-38
FLOAT data type 3-9
Floating point 3-9
Foreign key 2-27
Fragment
  adding 5-34
  altering 5-32, 5-33
  changing the number of 5-31
  description of 5-3
  dropping 5-35
  nonoverlapping fragments on multiple columns 5-19
  nonoverlapping fragments on single column 5-17
  when eliminated from search 5-15
FRAGMENT BY EXPRESSION clause, of CREATE TABLE statement 5-11
Fragment elimination
  in expression-based distribution scheme 5-22
  in hybrid distribution scheme 5-24
  in system-defined hash distribution scheme 5-23
Fragment expression, arbitrary expression 5-12
Fragmentation
  across coservers 5-6
  backup-and-restore operations and 5-7
  creating an explicit rowid column 5-41
  dbslice role in 5-6
  description of 5-3
  distribution schemes for 5-9
  equality search 5-16
  expressions, how evaluated 5-13
  fragment elimination 5-15
  goals of 5-6
  logging and 5-8
  modifying 5-30
  of table indexes 5-39
  of temporary tables 5-37
  range search 5-16
  reinitializing 5-31
  rowids and 5-41
  use of primary key 5-41
  with PDQ 5-6
Fragmented table
  accessing data 5-42
  adding a fragment 5-34
  created from multiple non-fragmented table 5-29
  creating 5-28
  creating from non-fragmented table 5-29
  dropping a fragment 5-35
  how to create 5-26
  modifying 5-30
  use of rowid 5-43
  using primary keys 5-42
Fragment-elimination behavior
  for OnLine database server 5-20
  for OnLine XPS database server 5-21
Functional dependency 2-34

G

Generalized-key index
  defining
    on a selection 7-20
    on an expression 7-21
    on joined tables 7-21
GK index. *See* Generalized-key index.
Global Language Support (GLS)
  description of Intro-5
  use of locales 1-10
GL_DATETIME environment variable 3-16
GRANT statement
  database-level privileges 8-5
  table-level privileges 8-7
Granularity, of fact table 6-11
GROUP BY keywords, restrictions in modifiable view 8-26

H

Hash distribution scheme. *See* System-defined hash distribution scheme.
High-Performance Loader (HPL), loading data with 7-9
Hybrid distribution scheme
  description of 5-4, 5-14

I

Icons
  comment Intro-9
  compliance Intro-11
  feature Intro-10
  platform Intro-10
  product Intro-10
Index
  attached, definition of 5-39
  bitmap, description of 7-18
  detached, definition of 5-40
  for data-warehousing environments 7-18
  fragmentation of table index 5-39
  generalized-key
    defining 7-20
    description of 7-18
Index privilege 8-10
Industry standards, compliance with Intro-16
INFORMIXDIR/bin directory Intro-5
INIT clause, of ALTER FRAGMENT 5-31
Insert privilege 8-8, 8-31
INSERT statement
  privilege for 8-5, 8-8
  with a view 8-28
INTEGER data type 3-7
INTERVAL data type
  description of 3-15
  display format 3-16
INTO TEMP keywords, restrictions in view 8-25
ISO 8859-1 code set Intro-5
Isolation level, default in ANSI-compliant database 1-8

J

Join, restrictions in modifiable view 8-26

K

Key
  composite 2-26
  foreign 2-27
  primary 2-25
Key column 2-25

L

Light appends, description of 7-15
Loading data
  with dbload utility 4-12
  with external tables 4-12
Locale Intro-5, 1-10
Logging
  buffered 4-6
  choosing for the database server 4-5
  unbuffered 4-6
Logging table
  characteristics of 7-14
  creation of 7-13

M

Machine notes Intro-14
Mandatory entity in relationship 2-10
Many-to-many relationship 2-10, 2-13, 2-29
Message file, error messages Intro-13
MODE ANSI keywords
  ANSI-compliant logging 4-6
  specifying ANSI compliance 1-6
MODIFY clause of ALTER FRAGMENT 5-32
Modifying a fragmentation strategy 5-30
Modifying fragmented tables 5-30
MONEY data type
  description of 3-11
  display format 3-12
  international money formats 3-12

N

NCHAR data type 3-17
NLS, compatibility with version 7.2 products 1-10
Nondecomposable attributes 2-17
Nonlogging table
  characteristics of 7-14
  creation of 7-13
Normal form 2-31
Normalization
  benefits 2-31
  first normal form 2-32
  of data model 2-31
  rules 2-31
  rules, summary 2-35
  second normal form 2-34
  third normal form 2-34
NOT NULL keywords, use in CREATE TABLE statement 4-6
Null value
  defined 3-23
  restrictions in primary key 2-25
NVARCHAR data type 3-18

O

ON-Archive, for backup-and-restore operations 5-7
ON-BAR, for backup-and-restore operations 5-7
One-to-many relationship 2-10, 2-13
One-to-one relationship 2-10, 2-13
On-line analytical processing (OLAP), description of 6-5
On-line manuals Intro-12
On-line transaction processing (OLTP), description of 6-5
Operational data store, description of 6-4
Operational table 7-17
Optional entity in relationship 2-10
ORDER BY keywords, restrictions in view 8-25
Ownership 8-7

P

PDQ (parallel database query), used with fragmentation 5-6
Performance, buffered logging 4-6
Platform icons Intro-10
Populating tables 4-12
Primary key
  definition of 2-25
  restrictions with 2-25
  use in fragmented table 5-42
Primary-key constraint, composite 2-26
Printed manuals Intro-13
Privilege
  ANSI-compliant databases and 1-7
  automating 8-13
  column-level 8-10
  Connect 8-5
  database-administrator 8-7
  database-level 8-5
  DBA 8-7
  Delete 8-8, 8-31
  encoded in system catalog 8-9
  Execute 8-12
  granting 8-5 to 8-14
  Index 8-10
  Insert 8-8, 8-31
  needed to create a view 8-30
  on a view 8-31
  procedure-level 8-12
  Resource 8-6
  Select 8-8, 8-10, 8-30
  table-level 8-8
  Update 8-8, 8-31
  views and 8-30 to 8-33
Procedure-level privileges 8-12
Product icons Intro-10
Program group
  Documentation notes Intro-15
  Release notes Intro-15
PUBLIC keyword, privilege granted to all users 8-6

R

Raw permanent table
  altering to 7-18
  description of 7-16
Recursive relationship 2-12, 2-30
Redundant relationship 2-31
References privilege 8-10
Referential integrity, defining
    primary and foreign keys 2-27
Related reading Intro-15
Relational model
  attribute 2-17
  description of 2-3
  entity 2-5
  many-to-many relationship 2-13
  normalizing data 2-31
  one-to-many relationship 2-13
  one-to-one relationship 2-13
  resolving relationships 2-29
  rules for defining tables, rows,
      and columns 2-23
Relationship
  attribute 2-17
  cardinality 2-15
  cardinality constraint 2-11
  complex 2-30
  connectivity 2-10, 2-13
  defining in data model 2-9
  entity 2-6
  existence dependency 2-10
  mandatory 2-10
  many-to-many 2-10, 2-13
  many-to-many, resolving 2-29
  one-to-many 2-10, 2-13
  one-to-one 2-10, 2-13
  optional 2-10
  recursive 2-30
  redundant 2-31
  using matrix to discover 2-11
Release notes
  location Intro-14
  program item Intro-15
Repository, description of 6-5
Resource privilege 8-6
REVOKE statement, granting
    privileges 8-5 to 8-14
Role
  creating with CREATE ROLE
      statement 8-15
  definition 8-14
  enabling with SET ROLE
      statement 8-16
  granting privileges with GRANT
      statement 8-16
  rules for naming 8-15
  sysroleauth table 8-17
  sysusers table 8-17
Round-robin distribution scheme
  and INSERT cursors 5-4
  and INSERT statements 5-4
  description of 5-4
  using 5-11
Row
  defining 2-23
  in relational model 2-23
Rowid
  creating in a fragmented
      table 5-43
  description of 5-41
  dropping in a fragmented
      table 5-43
  for fragmented table 5-41
  in fragmented tables 5-43

S

sales_demo database
  creating with command files 7-11
  data sources for 7-6
  using SQL statements
    to create 7-4
    to load 7-9
Sample-code conventions Intro-11
Scratch table 7-15
Second normal form 2-34
Security
  constraining inserted values 8-22,
      8-29
  database-level privileges 8-4
  making database inaccessible 8-4
  restricting access to
      columns 8-22, 8-23
  restricting access to rows 8-22,
      8-24
  restricting access to view 8-30
  table-level privileges 8-10
  using operating-system
      facilities 8-4
  with stored procedures 8-3
Select privilege
  column level 8-10
  definition of 8-8
  with a view 8-30
SELECT statement
  in modifiable view 8-26
  privilege for 8-5, 8-8
Semantic integrity 3-3
SERIAL data type 3-7
SET LOG statement, buffered vs.
    unbuffered 4-6
SMALLFLOAT data type 3-9
SMALLINT data type 3-7
Software dependencies Intro-4
Specifying ANSI compliance 1-6
Standard permanent table
  altering to 7-18
  description of 7-17
Star-join schema
  description of 6-10
  See also Dimensional data model.
Static table 7-16
Stored procedure
  granting privileges on 8-12
  security purposes 8-3
stores7 database Intro-5
Synonym
  chains 4-10
  in ANSI-compliant database 1-10
Synonyms for table names 4-8
sysfragments table 5-5
syssyntable system catalog
    table 4-9
System catalog
  privileges in 8-9
  syscolauth 8-9
  systabauth 8-9
  sysusers 8-9
  table, sysfragments 5-5
System-defined hash distribution
    scheme
  description of 5-4
  fragment elimination in 5-23
  using 5-13

T

Table
  candidate keys, defined 2-26
  composite key, defined 2-26
  creating a table 4-6
  descriptor column 2-25
  foreign key, defined 2-27
  in relational model 2-23
  key column 2-25
  loading data into 4-12
  names, synonyms 4-8
  ownership 8-7
  primary key in 2-25
  represents an entity 2-25
Table-index fragmentation 5-39
Table-level privileges
  access privileges 8-8
  Alter privilege 8-10
  definition and use 8-8
  Index privilege 8-10
  References privilege 8-10
Temp table 7-15
Temporary table
  and fragmentation 5-37
  explicit 5-37
  flexible 5-38
TEXT data type
  description of 3-21
  using 3-22
Third normal form 2-34
Transaction logging
  ANSI-compliant database, effects on 1-7
  buffered 4-6
  establishing with CREATE DATABASE statement 4-4
  turning off for faster loading 4-13
Transaction, ANSI-compliant database, effects on 1-6
Transitive dependency 2-34

U

Unbuffered logging 4-6
UNION keyword
  in a view definition 8-25
  restrictions in modifiable view 8-27
UNIQUE keyword
  constraint in CREATE TABLE statement 4-6
  restrictions in modifiable view 8-27
Update privilege
  definition of 8-8
  with a view 8-31
UPDATE statement
  applied to view 8-27
  privilege for 8-5, 8-8
Utility program
  dbload 4-13, 7-3, 7-4, 7-9, 7-13
  dbschema 4-11

V

VARCHAR data type 3-18
View
  creating 8-23
  deleting rows in 8-27
  description of 8-22
  dropped when basis is dropped 8-25
  effect of changing basis 8-25
  inserting rows in 8-28
  modifying 8-26 to 8-30
  null inserted in unexposed columns 8-28
  privilege when accessing 8-31
  privileges 8-30 to 8-33
  restrictions on modifying 8-26
  updating duplicate rows 8-28
  using WITH CHECK OPTION keywords 8-29
  virtual column 8-27

W

WHERE keyword, enforcing data constraints 8-30
WITH CHECK OPTION keywords, of CREATE VIEW statement 8-29
WITH ROWIDS clause, of CREATE TABLE statement 5-43

X

X/Open compliance, level Intro-16