Progress Database Design Guide

.

May 2001

Product Code: 4511
Item Number: 81079;9.1C

Contents

Figures

Tables

Procedures

Preface

Purpose

The *Progress Database Design Guide* introduces the fundamental principles of relational database design. Use this book if you are unfamiliar with relational database concepts, or if you need an advanced discussion of Progress 4GL index usage. Topics include table relationships, normalization principles, database design basics, and indexing.

Audience

This guide is for new-users of relational database management systems and application developers who are unfamiliar with relational database concepts.

Organization of This Manual

Chapter 1, "Introduction"

This chapter introduces computerized databases and the relational model. It describes the basic components of a relational database including: tables, columns (or fields), and rows (or records). It also includes a definition of the Progress sports database.

Chapter 2, "Table Relationships and Normalization"

This chapter describes the various relationships found in a relational database. It also introduces the concept of normalization and describes the first three normal forms.

Chapter 3, "Database Design Basics"

This chapter overviews the steps for analyzing and implementing a database design.

Chapter 4, “Defining Indexes”

This chapter introduces Progress indexes and how they work.

Chapter 5, “Progress 4GL Index Usage”

This chapter explains in detail the Progress 4GL algorithms for choosing indexes.

Chapter 6, “Progress 4GL Word Indexes”

This chapter introduces word indexes and how to use them in a Progress 4GL application.

Chapter 7, “Constraints and Indexes Using SQL-92”

This chapter explains what constraints and indexes are and why to define them.

Chapter 8, “Progress 4GL Triggers”

This chapter introduces 4GL triggers and how to use them.

Chapter 9, “Java Stored Procedures and Triggers”

This chapter introduces Java stored procedures and triggers and how to use them.

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples

- System output
- Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, GET, GO
  ALT, CTRL, SPACEBAR, TAB

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

```
FOR EACH Customer:
   DISPLAY Name.
END.
```

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

  In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

  SYNTAX

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

  In some instances, square brackets are not a syntax notation, but part of the language.

  For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

  SYNTAX

```
INITIAL [ constant [ , constant ] ... ]
```

  NOTE: The ellipsis (**. . .**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In this example, you must specify the items BY and *expression* and can optionally specify the item *DESCENDING*, in that order:

  SYNTAX

```
{ BY expression [ DESCENDING ] }
```

  In some cases, braces are not a syntax notation, but part of the language.

For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

SYNTAX

```
{ &argument-name }
```

- A vertical bar (**|**) indicates a choice.

  In this example, EACH, FIRST, and LAST are optional, but you can only choose one:

  SYNTAX

  ```
  PRESELECT [ EACH | FIRST | LAST ] record-phrase
  ```

  In this example, you must select one of *logical-name* or *alias*:

  SYNTAX

  ```
  CONNECTED ( { logical-name | alias } )
  ```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

  In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

  ```
  MAXIMUM ( expression , expression [ , expression ] ... )
  ```

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

  SYNTAX

  ```
  MESSAGE { expression | SKIP [ (n) ] } ...
  ```

In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

SYNTAX

```
{ include-file
    [ argument | &argument-name = "argument-value" ] ... }
```

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

  In this example, WITH is followed by several optional items:

  SYNTAX

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

  In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

  SYNTAX

```
ASSIGN {   { [ FRAME frame ]
               { field [ = expression ] }
               [ WHEN expression ]
           } ...
         | { record [ EXCEPT field ... ] }
       }
```

Progress Messages

Progress displays several types of messages to inform you of routine and unusual occurrences:

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify, or that are assumed, as part of the procedure. This is the most common action taken following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

Progress messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

Use Progress online help to get more information about Progress messages. On the Windows platform, many Progress tools include the following Help menu options to provide information about messages:

- Choose Help→ Recent Messages to display detailed descriptions of the most recent Progress message and all other messages returned in the current session.
- Choose Help→ Messages, then enter the message number to display a description of any Progress message. (If you encounter an error that terminates Progress, make a note of the message number before restarting.)
- In the Procedure Editor, press the **HELP** key (**F2** or **CTRL-W**).

On the UNIX platform, you can use the Progress PRO command to start a single-user mode character Progress client session and view a brief description of a message by providing its number. Follow these steps:

1 ♦ Start the Progress Procedure Editor:

```
install-dir/dlc/bin/pro
```

2 ♦ Press **F3** to access the menu bar, then choose Help→ Messages.

3 ♦ Type the message number, and press **ENTER**. Details about that message number appear.

4 ♦ Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose File→ Exit.

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows **and** Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A guide that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Progress Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Language Tutorial for Windows and *Progress Language Tutorial for Character*

Platform-specific tutorials designed for new Progress users. The tutorials use a step-by-step approach to explore the Progress application development environment using the 4GL.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Progress Basic Development Tools (Character only; information for Windows is in online help)

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for UNIX*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Building Distributed Applications Using the Progress AppServer

A guide that provides comprehensive information about building and implementing distributed applications using the Progress AppServer. Topics include basic product information and terminology, design options and issues, setup and maintenance considerations, 4GL programming details, and remote debugging.

Progress External Program Interfaces

A guide to accessing non-Progress applications from Progress. This guide describes how to use system clipboards, UNIX named pipes, Windows dynamic link libraries, Windows dynamic data exchange, Windows ActiveX controls, and the Progress Host Language Call Interface to communicate with non-Progress applications and extend Progress functionality.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

Database

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial. Each DataServer has its own guide, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product Guide*.

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version 3.1 or higher installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: `http://www.adobe.com/prodindex/acrobat/readstep.html`.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress Open Client Developer's Guide

A guide that describes how to write and deploy Java and ActiveX applications that run as clients of the Progress AppServer. The guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Developer's Guide

Provides a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

WebSpeed Product Update Bulletin

A booklet that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Welcome to WebSpeed! (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

Pocket WebSpeed (Hard copy only)

A reference that lets you quickly look up information about the SpeedScript language or the WebSpeed programming environment.

SQL-92 Reference

(These are non-Progress resources available from your technical bookseller.)

A Guide to the SQL Standard

Date, C.J., with Hugh Darwen. 1997. Reading, MA: Addison Wesley.

Understanding the New SQL: A Complete Guide

Melton, Jim (Digital Equipment Corporation) and Alan R. Simon. 1993. San Francisco: Morgan Kaufmann Publishers.

1

Introduction

This chapter briefly defines these topics:

- What a database is
- Benefits of a computerized database
- The relational model

The purpose of this chapter is to provide an introduction to the relational database. It is not meant to provide an exhaustive resource on this topic; it is merely intended to help you get started.

1.1 What Is a Database?

A *database* is a collection of data that you can search through in a systematic way to maintain and retrieve information. A database can be computerized or noncomputerized. Some noncomputerized databases that you're familiar with are a telephone book, a filing cabinet, and a library card catalog system. To retrieve information from each of these databases, you proceed accordingly:

- To look up a friend's phone number, you thumb through the telephone book of the town they live in, and search for their last name, followed by their first name. If there is more than one occurrence of the name, you check the addresses, and by process of elimination, you determine your friend's phone number. The telephone book has a very simple and restricted structure. You cannot, for instance, look up a phone number by the person's first name or by their address.

- To check the balance on a customer's account, you rifle through a filing cabinet to locate the customer's folder and pull out the piece of paper with the current balance on it. This database, too, has a restricted structure. You cannot, for instance, look up a customer's record by their address or the name of their sales representative. Furthermore, to look up all customers with an outstanding balance of more than $1,000, you must go through all the folders individually to find the customers that meet this criteria.

- To find a book in the library, you must first determine whether the library has the book by looking it up in the card catalogs, either by title, author's name, or subject matter. You note its decimal ID number, then go to the appropriate shelf and locate the book. Of the three databases, the card catalog is the most sophisticated because it allows you to look up a book in at least three different ways.

To summarize, if you want to locate information quickly and effortlessly in each of these noncomputerized databases, you must store every piece of data—name, customer folder, or catalog card—in some sort of order. Even then, it can take anywhere from a few minutes to several hours to locate the data, depending on the size of your database and the complexity of the query.

1.2 Computerized Databases

Let's take a closer look at the example of the filing cabinet database. When a customer calls you to place an order, you go through a series of steps. First, you pull out the customer's file from the customer cabinet to determine whether the customer is a current account. Then you rummage through the inventory cabinet and pull out the appropriate item files to see whether you have the ordered items in stock. After that, you fill out an order form by listing each item, its price, and the grand total for the order. Finally, you make appropriate changes to the item files in inventory to reflect the current quantity.

Imagine how tedious and unmanageable these repetitive tasks can become if you have several hundred customers calling you each day. In this situation, automating your database makes a lot of sense. A computerized database offers you many advantages, including:

- **Centralized and shared data** — You enter and store all your data in the computer. This minimizes the use of paper, files, folders, as well as the likelihood of losing or misplacing them. Once the data is in the computer, many users can access it via a computer network. The users' physical or geographical locations are no longer a constraint.

- **Current data** — Since users can quickly update data, the data available is current and ready to use.

- **Speed and productivity** — You can search, sort, retrieve, make changes, and print your data, as well as tally up the totals more quickly than performing these tasks by hand.

- **Accuracy and consistency** — You can design your database to validate data entry, thus ensuring that it is consistent and valid. For example, if a user enters OD instead of OH for Ohio, your database can display an error message. It can also ensure that the user is unable to delete a customer record that has an outstanding order.

- **Analysis** — Databases can store, track, and process large volumes of data from diverse sources. You can use the data collected from varied sources to track the performance of an area of business for analysis or to reveal business trends. For example, a clothes retailer can track faulty suppliers, customers' credit ratings, and returns of defective clothing. An auto manufacturer can track assembly-line operation costs, product reliability, and worker productivity.

- **Security** — You can protect your database by establishing a list of authorized user identifications and passwords. The security ensures that the user can perform only the operations that you permit. For example, you may allow the user to read data in your database but not allow them to update or delete it.

- **Crash recovery** — System failures are inevitable. With a database, data integrity is assured in the event of a failure. The database management system uses a transaction log to ensure that your data will be properly recovered when you restart after a crash.

- **Transactions** — The transaction concept provides a generalized error recovery mechanism to deal with the consequences of unexpected errors. Transactions ensure that a group of related database changes always occur as a unit; either all the changes are made or none of the changes are made. This allows you to restore the previous state of the database should an error occur after you have begun making changes.

Now that you understand the benefits of a computerized database system, let's take a look at the elements of relational databases.

1.3 Elements of a Relational Database

Relational databases are based on the relational model. The *relational model* is a group of rules set forth by E. F. Codd based on mathematical principles (relational algebra); it defines how database management systems should function. The basic structures of a relational database (as defined by the relational model) are tables, columns (or fields), rows (or records), and keys. This section describes these elements.

1.3.1 Tables

A *table* is a collection of logically related information treated as a unit. Figure 1–1 shows an example of the contents of a Customer table.

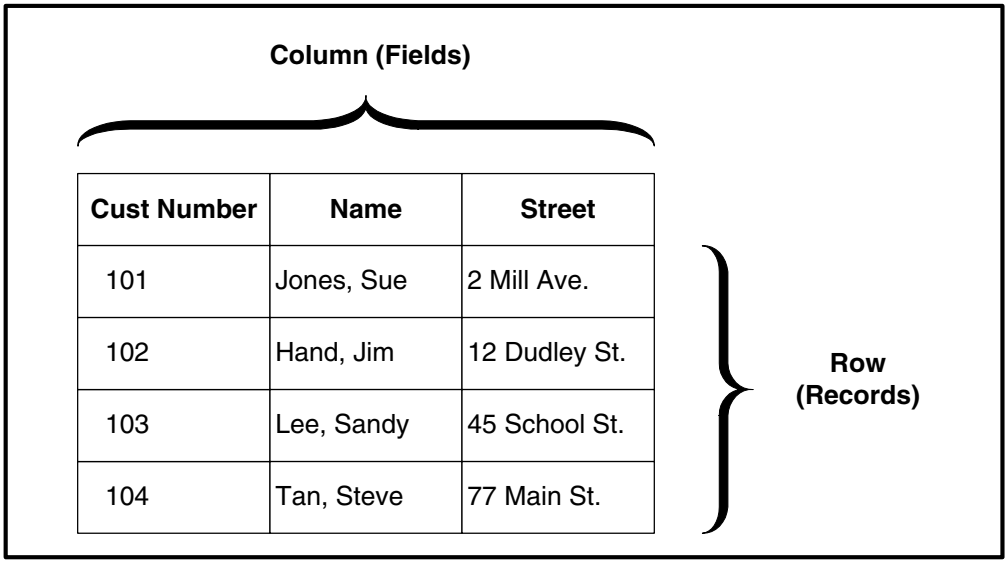


Figure 1–1: Columns and Rows in the Customer Table

Examples of other tables include an Order table that keeps track of the orders each customer places, an Assignment table that keeps track of all the projects each employee works on, and a Student Schedule table that keeps track of all the courses each student takes.

1.3.2 Rows

A table is made up of rows (or records). A *row* is a single occurrence of the data contained in a table. Each row is treated as a single unit. In the Customer table shown in Figure 1–1, there are four rows, and each row contains information about an individual customer.

Similarly, each row in the Order table represents an order that a customer places, a row in the Assignment table represents a project an employee works on, and a row in the Student Schedule table represents a course a student takes.

1.3.3 Columns

Rows are organized as a set of *columns* (or fields). All rows in a table comprise the same set of columns. In the Customer table, the columns are Cust Number, Name, and Street.

1.3.4 Keys

There are two types of keys: primary and foreign. A *primary key* is a column (or group of columns) whose value uniquely identifies each row in a table. Because the key value is always unique, you can use it to detect and prevent duplicate rows. A good primary key has these characteristics:

- It is **mandatory**; that is, it must store non-null values. If the column is left blank, duplicate rows can occur.
- It is **unique**. For example, the social security column in an Employee or Student table is a good key because it uniquely identifies each individual. The Cust Number column in the Customer table uniquely identifies each customer. It is less practical to use a person's name because more than one customer might have the same name. Also, databases do not detect variations in names as duplicates (for example, Cathy for Catherine, Joe for Joseph). Furthermore, people do sometimes change their names (for example, through a marriage or divorce).
- It is **stable**; that is, it is unlikely to change. As in the previous example, the social security number is a good key not only because it uniquely identifies each individual, but it is also unlikely to change, while a person's or customer's name might change.
- It is **short**; that is, it has few characters. Smaller columns occupy less storage space, database searches are faster, and entries are less prone to mistakes. For example, a social security column of 9 digits is easier to access than a name column of 30 characters.

NOTE: You can also have non-unique keys and word indexes.

A *foreign key* is a column value in one table that is required to match the column value of the primary key in another table. In other words, it is the reference by one table to another. If the foreign key value is not null, then the primary key value in the referenced table must exist. It is this relationship of a column in one table to a column in another table that provides the relational database with its ability to join tables. Chapter 2, "Table Relationships and Normalization" describes this concept in more detail.

A *composite key* is a key composed of multiple columns.

Indexes

An *index* in a database operates like the index tab on a file folder. It points out one identifying column, such as a customer's name, that makes it easier and faster to find the information you want.

When you use index tabs in a file folder, you use those pieces of information to organize your files. If you index by customer name, you organize your files alphabetically. If you index by

customer number, you organize them numerically. Indexes in the database serve the same purpose.

You may use a single column to define a simple index, or a combination of columns to define a compound index. To decide which columns to use, you determine how the data in the table will be accessed. If users frequently look up customers by last name, then the last name is a candidate for an index. It is typical to base indexes on primary keys (columns that contain unique information).

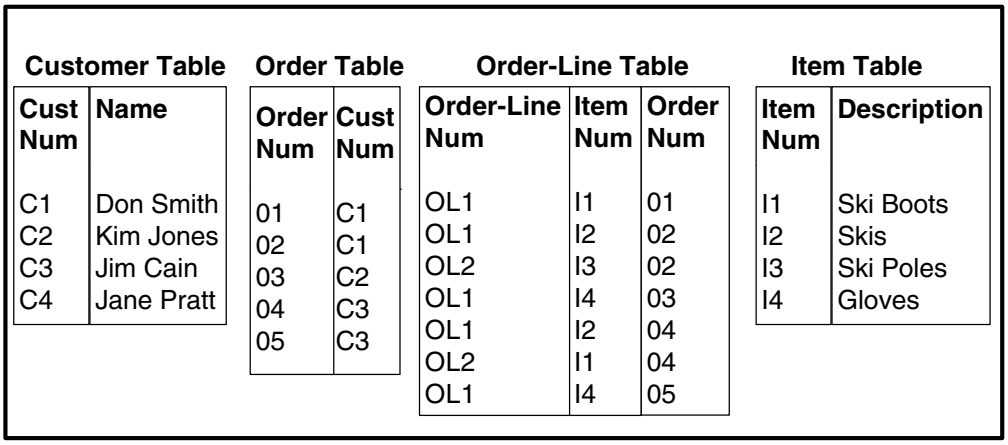
An index has these advantages:

- Faster row search and retrieval. It is more efficient to locate a row by searching a sorted index table than by searching an unsorted table.
- In a 4GL implementation, records are ordered automatically to support your particular data access patterns. No matter how you change the table, when you browse or print it, the rows appear in indexed order instead of their stored physical order on disk.
- When you define an index as unique, each row is unique. This ensures that duplicate rows do not occur. A unique index may contain nulls. However, a primary key, although unique, may not contain nulls.
- A combination of columns can be indexed together to allow you to sort a table in several different ways at once (for example, sort the Projects table by a combined employee **and** date column).
- Efficient access to data in multiple related tables.
- When you design an index as unique, each key value must be unique.

1.3.5 Applying the Principles of the Relational Model

The relational model organizes data in tables and lets you create relationships among tables by referencing columns that are common to both—the primary and foreign keys. It is easiest to understand this concept of relationships between tables with a common business example.

Many businesses need to track information about customers and their orders. So their database probably includes a Customer table and that Customer table might include the Customer Number, Name, Sales Representative, and Postal Code. They might want to uniquely identify each customer, so every customer in the Customer table has a unique Customer Number; every item in the Item table has a unique Item Number. These columns are the primary key columns in each of the named tables. Figure 1–2 shows these four tables.



Customer Table

Cust Num	Name
C1	Don Smith
C2	Kim Jones
C3	Jim Cain
C4	Jane Pratt

Order Table

Order Num	Cust Num
01	C1
02	C1
03	C2
04	C3
05	C3

Order-Line Table

Order-Line Num	Item Num	Order Num
OL1	I1	01
OL1	I2	02
OL2	I3	02
OL1	I4	03
OL1	I2	04
OL2	I1	04
OL1	I4	05

Item Table

Item Num	Description
I1	Ski Boots
I2	Skis
I3	Ski Poles
I4	Gloves

Figure 1–2: Example of a Relational Database

These are the tables described in Figure 1–2:

- The Customer table shows four rows, one for each individual customer. Each row has two columns: Cust Num and Name. Each column contains exactly one data value, such as C3 and Jim Cain. The primary key is Cust Num.
- The Order table shows five rows for the orders placed by the customers in the Customer table. Each Order row contains two columns: Cust Num (from the Customer table) and Order Num. The primary key is Order Num. The Cust Num column is the foreign key that relates the two tables. This relationship lets you find all the orders placed by a particular customer, as well as information about a customer for a particular order.
- The Order-Line table shows seven rows for the order-lines of each order. Each order-line row contains three columns: Order Num (from the Order table), Order-Line Num, and Item Num (from the Item table). The primary key is the combination of Order Num, Order Line. The two foreign keys (Order Num and Item Num) relate the Customer, Order, and Item tables so that you can find the following information:
  - All the order-lines for an order
  - Information about the order for a particular order-line
  - The item in each order-line
  - Information about an item
- The Item table shows four rows for each separate item. Each Item row contains two columns: Item Num and Description. Item Num is the primary key.

You want to find out which customers ordered ski boots. To gather this data from your database, you must know what item number identifies ski boots and who ordered them. There is no direct relationship between the Item table and the Customer table, so to gather the data you need, you join four tables (using their primary/foreign key relationships). First you relate the Items to Orders (through Order-Lines) and then the Orders to Customers. Figure 1–3 shows how this works.

NOTE: The figures show the primary key values character data for clarity. A numeric key is better and more efficient.

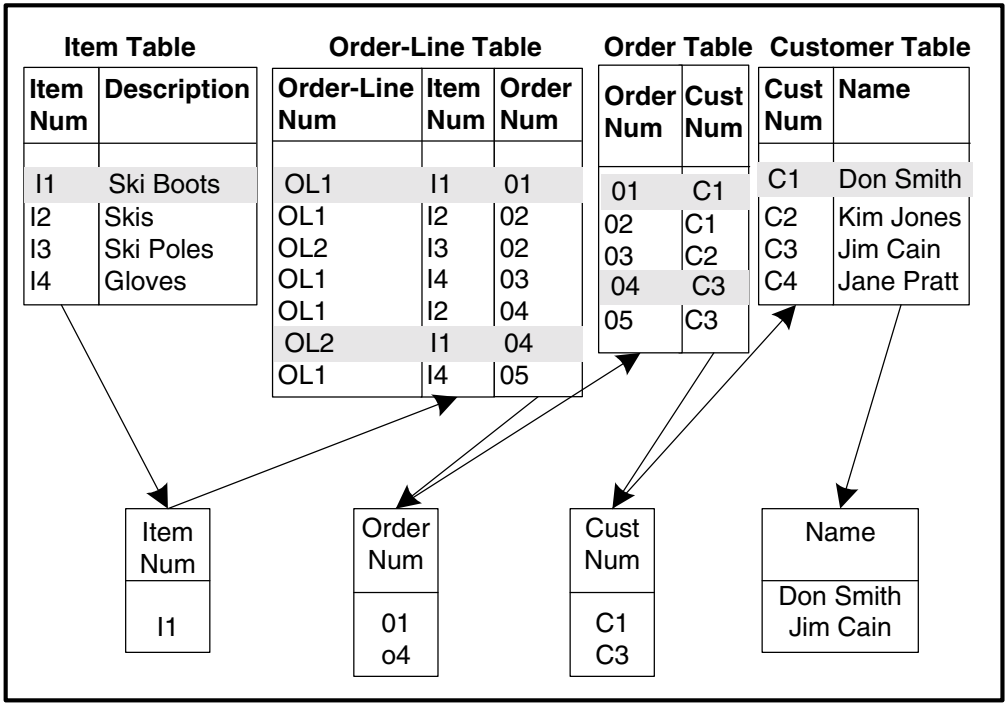


Figure 1–3: Selecting Records from Related Tables

1 ♦ First, select the Item table row whose Description value equals ski boots. The Item Number value is I1.

2 ♦ Next, you want to know the Orders that contain Item I1. But the Order table doesn't contain Items. It contains Order-Lines. So you first select the Order-Lines that contain I1, and determine the Orders related to these Order-Lines. Orders 01 and 04 contain Item Number I1.

3 ♦ Now that you know the Order Numbers, you can find out the customers who placed the orders. Select the 01 and 04 orders—determine the associated customer numbers. They are C1 and C3.

4 ♦ Finally, to find out the names of Customers C1 and C3, you select the Customer table rows that contain customer numbers C1 and C3. Don Smith and Jim Cain—ordered ski boots.

By organizing your data into tables and relating the tables with common columns, you can perform powerful queries. The structures of tables and columns are relatively simple to implement and modify, and the data is consistent regardless of the queries or applications used to access the data.

1.3.6 The Progress Database and the Relational Model

The Progress database is a relational database management system (RDBMS).You can add, change, manipulate, or delete the data and data structures in your database as your requirements change.

Database Schema and Metaschema

The physical structure of the Progress database consists of the elements of a relational database that you just read about: tables, columns, and indexes. The description of the database's structure—the files it contains, the columns within the files, views, etc.—is called the database *schema* or the *data definitions*.

The underlying structure of a database that makes it possible to store and retrieve data is called the *metaschema*. That is, the metaschema defines that there can be database files and columns and the structural characteristics of those database parts. All metaschema table names begin with an underscore (_). The metaschema is called the metaschema because it is data about the data.

NOTE: The metaschema is a set of tables that includes itself. Therefore, you can do ordinary queries on the metaschema to examine all table and index definitions, including the definitions of the metaschema itself.

The sports Database

The sports database is one of several sample databases provided with the product; it is frequently used in the documentation to illustrate database concepts and programming techniques. Table 1–1 describes the contents of the sports database.

Table 1–1: The Sports Database

Table	Description
Customer	Name, address, telephone, credit, and sales information for a specific customer
Invoice	Financial information by invoice for the receivables subsystem
Item	Stocking, pricing, and descriptive information about items in inventory
Local-Default	Format and label information for various countries
Order	Sales and shipping information for orders
Order-Line	Identification of and pricing information for a specific item ordered on a specific order
Ref-Call	Call history for a customer.
Salesrep	Names, regions, and quotas for the sales people
State	U.S. state names and their abbreviations

This database holds the information necessary to track customers, take and process orders, bill customers, and track inventory.

1.4 Key Points to Remember

- A database is an electronic filing system for organizing and storing data that relates to a broad subject area, such as sales and inventory.
- A database is made up of tables. A table is a collection of rows about a specific subject, such as customers.
- A row is a collection of pieces of information about one thing, such as a specific customer.

- A column is a specific item of information, such as a customer name.
- An index is a set of pointers to rows that you use as the basis for searching, sorting, or otherwise processing rows like customer number.
- A primary key is a column (or group of columns) whose value uniquely identifies each row in a table. Because the key value is always unique, you can use it to detect and prevent duplicate rows. It may not contain null data.
- An index in a database operates like the index tab on a file folder. It makes it easier to find information.
- A foreign key is a column (or group of columns) in one table whose values are required to match the value of a primary key in another table.

2

Table Relationships and Normalization

This chapter briefly defines these topics:

- Table relationships
- One-to-one, one-to-many, and many-to-many relationships
- Normalization theory
- First, second and third normal form

2.1 Table Relationships

In a relational database, tables relate to one another by sharing a common column or columns. This column, existing in two or more tables, allows the tables to be *joined*. When you design your database, you define the table relationships based on the rules of your business. (The relationship is frequently between primary and foreign key columns; however, tables can also be related by other non-key columns.)

In Figure 2–1, the Customer and Order tables are related by a foreign key, Customer Number.

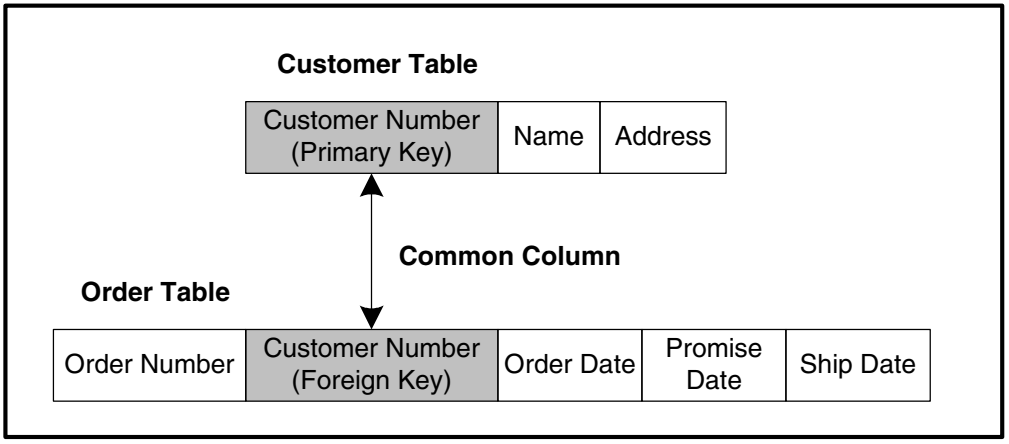


Figure 2–1: Relating the Customer and Order Tables

If the Customer Number is an index in both tables, you can do the following very quickly:

- Find all the orders for a given customer, and query information (such as order date, promised delivery date, the actual shipping date) for each order.
- Find customer information (such as name and address) for each order using an order's customer number.

NOTES

- In an SQL implementation, it is generally more efficient if your DATE, TIME and TIMESTAMP keys use the small exact numeric data types such as BIGINT, INT, SMALLINT and TINYINT.
- In an SQL implementation, for the larger non-numeric keys, including BINARY and VARBINARY, use the data types NUMERIC, CHAR and VARCHAR. However, it may be advantageous to map these to a small exact numeric data type. Do not use inexact numeric data types such as REAL and FLOAT since the results will be non-deterministic.

Figure 2–2 shows the relationship between the Customer and the Order tables.

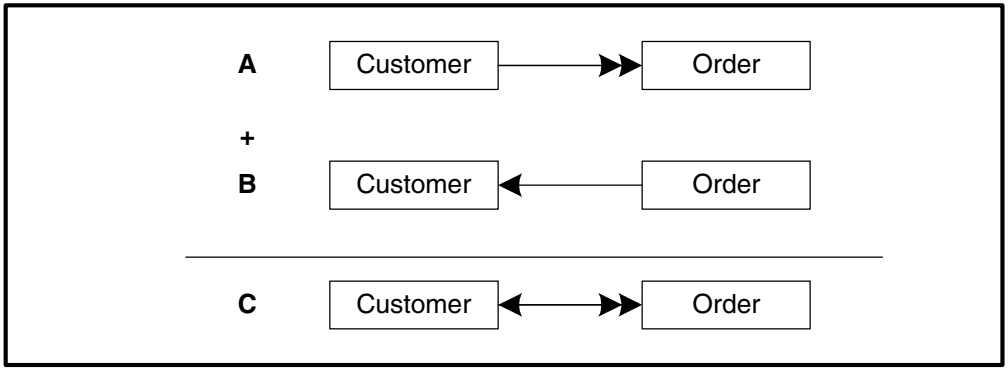


Figure 2–2: Relationship Between the Customer and Order Tables

From A, the Customer's view point, the relationship from Customer to Order is one-to-many because one customer can have many orders (as shown by the double arrows).

From B, the Order's view point, the relationship from Order to Customer is one-to-one because one order corresponds to exactly one customer (as shown by the single arrow).

C is the summary of the relationships from Customer to Order and from Order to Customer.

The following sections explain the three types of table relationships—one-to-one, one-to-many, and many-to-many-in greater depth.

2.1.1 One-to-one Relationship

A *one-to-one* relationship exists when each row in one table has only one related row in a second table. For example, a business may decide to assign one office to exactly one employee. Thus, one employee can have only one office. The same business may also decide that a department can have only one manager. Thus, one manager can manage only one department. Figure 2–3 shows this relationship.

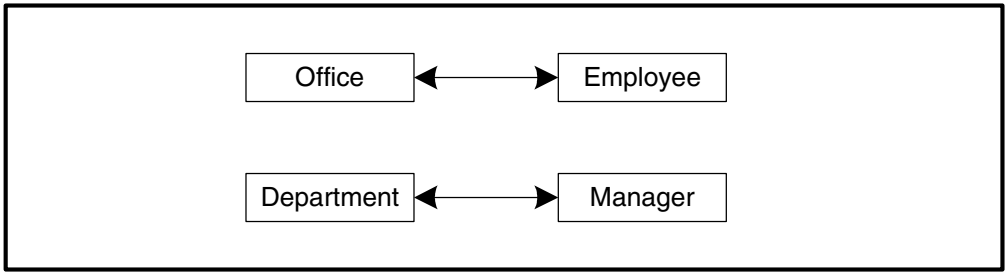


Figure 2–3: Examples of a One-to-one Relationship

However, the business may also decide that for one office there is zero or one employee; for one department there is no manager or there is one manager then the relationship is described as a zero-or-one relationship.

2.1.2 One-to-many Relationship

A *one-to-many* relationship exists when each row in one table has one or many related rows in a second table. Figure 2–4 shows some examples: one customer can place many orders. A sales representative can have many customer accounts.

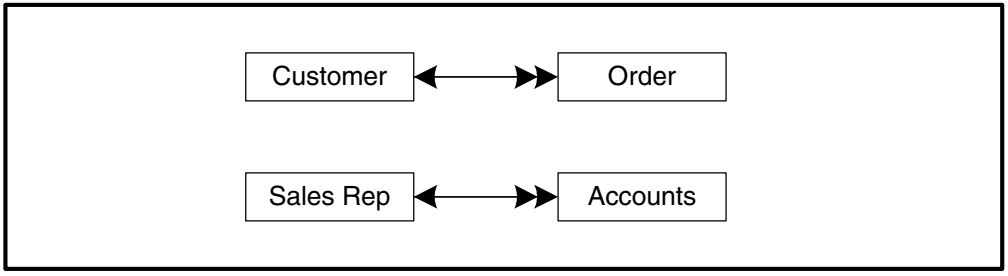


Figure 2–4: Examples of a One-to-many Relationship

However, the business rule may be that for one customer there may be zero-or-many orders, one student can take zero-or-many courses, and a sales representative may have zero-or-many customer accounts. This relationship is described as a zero-or-many relationship.

2.1.3 Many-to-many Relationship

A *many-to-many* relationship exists when a row in one table has many related rows in a second table. Likewise, those related rows have many rows in the first table. Figure 2–5 shows some examples: an order can contain many items, and an item can appear in many different orders; an employee can work on many projects, and a project can have many employees working on it.

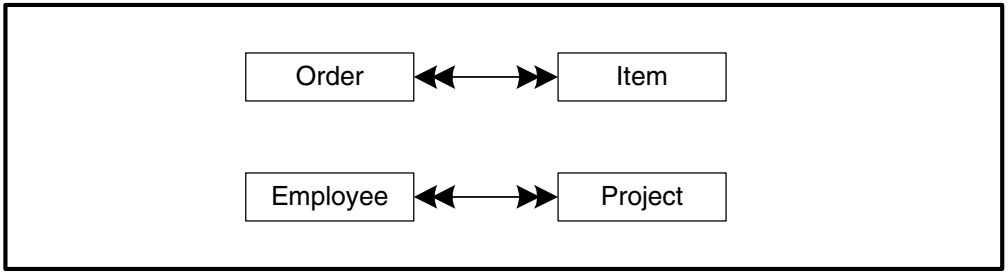


Figure 2–5: Examples of the Many-to-many Relationship

Accessing information in tables with a many-to-many relationship is difficult and time consuming. For efficient processing, you can convert the many-to-many relationship tables to **two** one-to-many relationships by connecting these two tables with a cross-reference table that contains the related columns.

For example, to establish a one-to-many relationship between Order and Item tables, create a cross-reference table Order-Line as shown in Figure 2–6. The Order-Line table contains both the Order Number and the Item Number. Without this table, you would have to store repetitive information or create multiple columns in both the Order and Item tables.

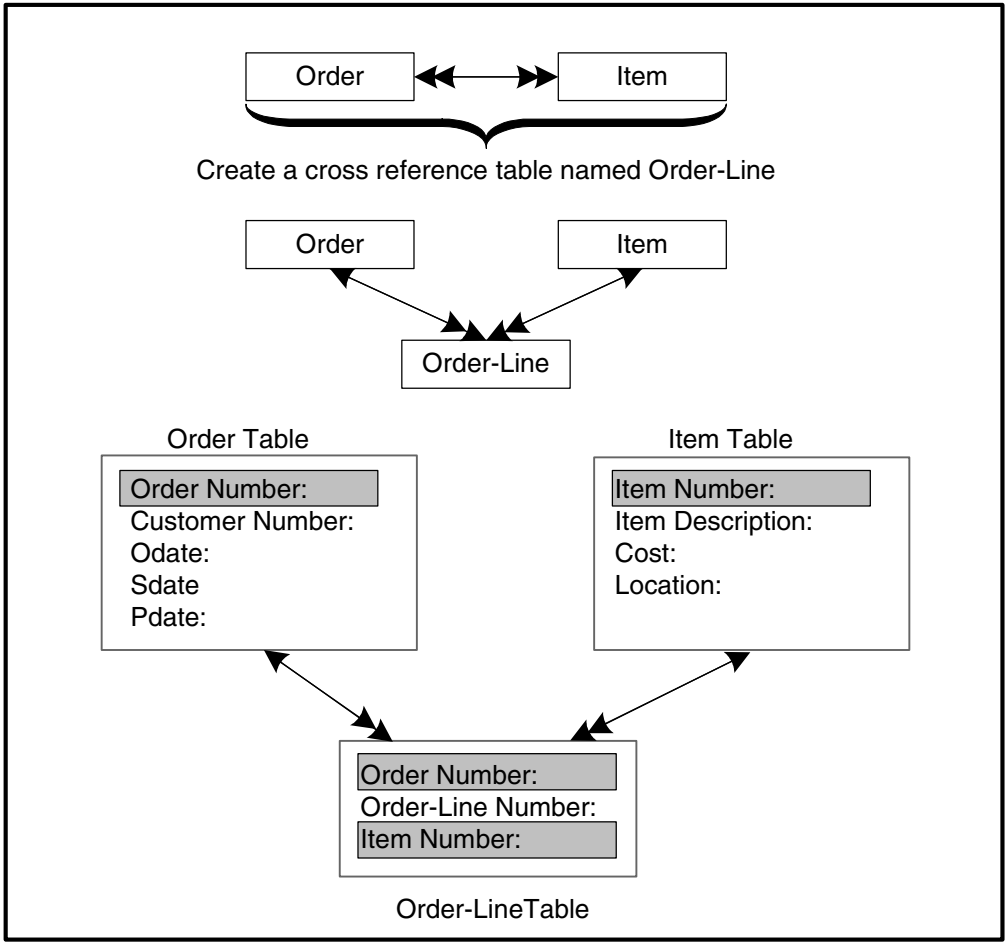


Figure 2–6: Using a Cross-reference Table to Relate Order and Item Tables

2.2 Normalization

This section provides an overview of normalization and the first three normal forms. *Normalization* is an iterative process during which you streamline your database to reduce redundancy and increase stability. During the normalization process you determine in which table a particular piece of data belongs based on the data itself—its meaning to your business and its relationship to other data. Your database design results in a data-driven, not processor application-driven design. This results in a database implementation that is more stable over time.

Normalization does not require you to have advanced database skills. It does require that you know your business and know the different ways you want to relate the data in your business. When you normalize your database, you eliminate columns that:

- Contain more than one value
- Duplicate or repeat
- Do not describe the table
- Contain redundant data
- Can be derived from other columns

The following sections describe the rules for the first, second and third normal form

2.2.1 The First Normal Form

The columns of a table in first normal form have these characteristics:

- They contain only one value
- They occur once (that is, they do not repeat)

The first rule of normalization is that you must remove duplicate columns or columns that contain more than one value to a new table.

First, examine an unnormalized Customer table, as shown in Table 2–1.

Table 2–1: Unnormalized Customer Table with Several Values in a Column

Cust Num	Name	Street	Order Number
101	Jones, Sue	2 Mill Ave.	M31, M98, M129
102	Hand, Jim	12 Dudley St.	M56
103	Lee, Sandy	45 School St.	M37, M40
104	Tan, Steve	67 Main St.	M41

Here, the Order Number column has more than one entry. This makes it very difficult to perform even the simplest tasks—for example, delete all order numbers M56 and above, find the total number of orders for a customer, or print orders in sorted order. To perform any of those tasks you need a complex algorithm to examine each value in the Order Number column for each row. It is, therefore, crucial that each column in a table consists of exactly one value.

Table 2–2 shows the same Customer table in a different unnormalized format.

Table 2–2: Unnormalized Table with Multiple Duplicate Columns

Cust Num	Name	Street	Order Number1	Order Number2	Order Number3
101	Jones, Sue	2 Mill Ave.	M31	M98	M129
102	Hand, Jim	12 Dudley St.	M56	Null	Null
103	Lee, Sandy	45 School St.	M37	M140	Null
104	Tan, Steve	67 Main St.	M41	Null	Null

Here, instead of a single Order Number column, there are three separate but duplicate columns for Order Number (Order Number1, Order Number2, and Order Number3). This format is also not efficient. What happens if a customer has more than three orders? You must either add a new column or clear an existing column value to make a new entry. It is difficult to estimate a reasonable maximum number of orders for a customer. If your business is brisk, you might have to create 200 Order Number columns for a row. Also if a customer has only 10 orders, the database will contain 190 null values for this customer.

Furthermore, it is difficult and time consuming to retrieve data with repeating columns. For example, to determine which customer has Order Number M98, you must look at each Order Number column individually (all 200 or more of them) in every row to find a match.

To reduce the Customer table to the first normal form, split it into two smaller tables, one table to store only Customer information (see Table 2–3) and another to store only Order information (see Table 2–4).

Table 2–3: Customer Table

Cust Num (Primary Key)	Name	Street
101	Jones, Sue	2 Mill Ave.
102	Hand, Jim	12 Dudley St.
103	Lee, Sandy	45 School St.
104	Tan, Steve	67 Main St.

Table 2–4: Order Table

Order Number (Primary Key)	Cust Num (Foreign Key)
M31	101
M98	101
M129	101
M56	102
M37	103
M140	103
M41	104

Note that there is only one instance of a column in the Customer and Order tables and each column contains exactly one value. The Cust Num column in the Order table relates to the Cust Num column in the Customer table.

A table that is normalized to the first normal form has these advantages:

- It allows you to create any number of orders for each customer without having to add new columns
- It allows you to query and sort data for orders very quickly because you search only one column—Order Number
- It uses disk space more efficiently—no empty columns are stored

2.2.2 The Second Normal Form

A table is in the second normal form when it is in the first normal form and only contains columns that give you information about the key of the table.

The second rule of normalization is that you must remove to a new table those columns that don't depend on the primary key of the current table.

Table 2–5 shows a Customer table that is in first normal form because there are no duplicate columns and every column has exactly one value.

Table 2–5: Customer Table with Repeated Data

Cust Num	Name	Street	Order Number	Order Date	Order Amount
101	Jones, Sue	2 Mill Ave.	M31	3/19/91	$400.87
101	Jones, Sue	2 Mill Ave.	M98	8/13/91	$3,000.90
101	Jones, Sue	2 Mill Ave.	M129	2/9/91	$919.45
102	Hand, Jim	12 Dudley St.	M56	5/14/90	$1,000.50
103	Lee, Sandy	45 School St.	M37	12/25/90	$299.89
103	Lee, Sandy	45 School St.	M140	3/15/91	$299.89
104	Tan, Steve	67 Main St.	M41	4/2/90	$2,300.56

However, the table is not normalized to the second rule. It has these problems:

- First, note that the first three rows in this table repeat the same data for the columns: Cust Num, Name, and Street. This is *redundant data*.
- Second, if the customer, Sue Jones, changes her address, you must then update all existing rows to reflect the new address. In this case, three rows. Any rows with the old address left unchanged leads to *inconsistent data*. Thus, your database lacks *integrity* .
- Third, you might want to trim your database by eliminating all orders before November 1, 1990, but in the process, you also lose all the customer information for Jim Hand and Steve Tan. The unintentional loss of rows during an update operation is called an *anomaly*.

So, how do you resolve the problems? Note that this table contains information about an individual customer, such as Cust Num, Name, and Street, that remain the same when you add an order. In other words, columns like Order Num, Order Date, and Order Amount do not pertain to the customer and do not depend on the primary key Cust Num. They should be in a different table.

To reduce the Customer table to the second normal form, divide it into two tables, as shown in Table 2–6 and Table 2–7.

Table 2–6: Customer Table

Cust Num (Primary Key)	Name	Street
101	Jones, Sue	2 Mill Ave.
102	Hand, Jim	12 Dudley St.
103	Lee, Sandy	45 School St.
104	Tan, Steve	67 Main St.

Table 2–7: Order Table *(1 of 2)*

Order Number (Primary Key)	Order Date	Order Amount	Cust Num (Foreign Key)
M31	3/19/91	$400.87	101
M98	8/13/91	$3,000.90	101

Table 2–7: Order Table *(2 of 2)*

Order Number (Primary Key)	Order Date	Order Amount	Cust Num (Foreign Key)
M129	2/9/91	$919.45	101
M56	5/14/90	$1,000.50	102
M37	12/25/90	$299.89	103
M140	3/15/91	$299.89	103
M41	4/2/90	$2,300.56	104

Note that the Customer table now contains only one row for each individual customer, while the Order table contains one row for every order, and Order Number is its primary key. The Order table contains a common column, Cust Num, that relates the Order rows with the Customer rows.

A table that is normalized to the second normal form has these advantages:

- It allows you to make updates to customer information in just one row
- It allows you to delete customer orders without eliminating necessary customer information
- It uses disk space more efficiently—no repeating or redundant data is stored

2.2.3 The Third Normal Form

A table is in the third normal form when it contains only independent columns, that is, columns not derived from other columns.

The third rule of normalization is that you must remove columns that can be derived from existing columns.

Table 2–8 shows an Order table with a Total After Tax column that is calculated from adding a 10% tax to the Order Amount column.

Table 2–8: Order Table with Derived Column

Order Number (Primary Key)	Order Date	Order Amount	Total After Tax	Cust Num (Foreign Key)
M31	3/19/91	$400.87	$441.74	101
M98	8/13/91	$3,000.90	$3,300.99	101
M129	2/9/91	$919.45	$1011.39	101
M56	5/14/90	$1,000.50	$1,100.55	102
M37	12/25/90	$299.89	$329.87	103
M140	3/15/90	$299.89	$329.87	103
M41	4/2/90	$2,300.56	$2,530.61	104

To reduce this table to third normal form, eliminate the Total After Tax column because it is a dependent column that changes when the Order Amount or tax change. For your report, you can create an algorithm to obtain the amount for Total After Tax. You need only keep the source value because you can always derive dependent values. Similarly, if you have an Employee table, you do not have to include an Age column if you already have a Date of Birth column, because you can always calculate the age from the date of birth.

A table that is in third normal form gives you these advantages:

- It uses disk space more efficiently—no unnecessary data is stored
- It contains only the necessary columns—superfluous columns are removed

Although a database normalized to the third normal form is desirable because it provides a high level of consistency, it may impact performance when you physically implement the database. When this occurs, consider denormalizing these tables. Chapter 3, “Database Design Basics” discusses denormalization.

3

Database Design Basics

Once you understand the basic structure of a relational database, you can begin the database design process. Designing a database is an iterative process that involves developing and refining a database structure based on the information and processing requirements of your business. This chapter overviews each phase of the design process. The phases are:

- Data analysis
- Logical database design
- Physical database design
- Physical implementation

3.1 Database Design Cycle

Figure 3–1 illustrates the database design cycle.

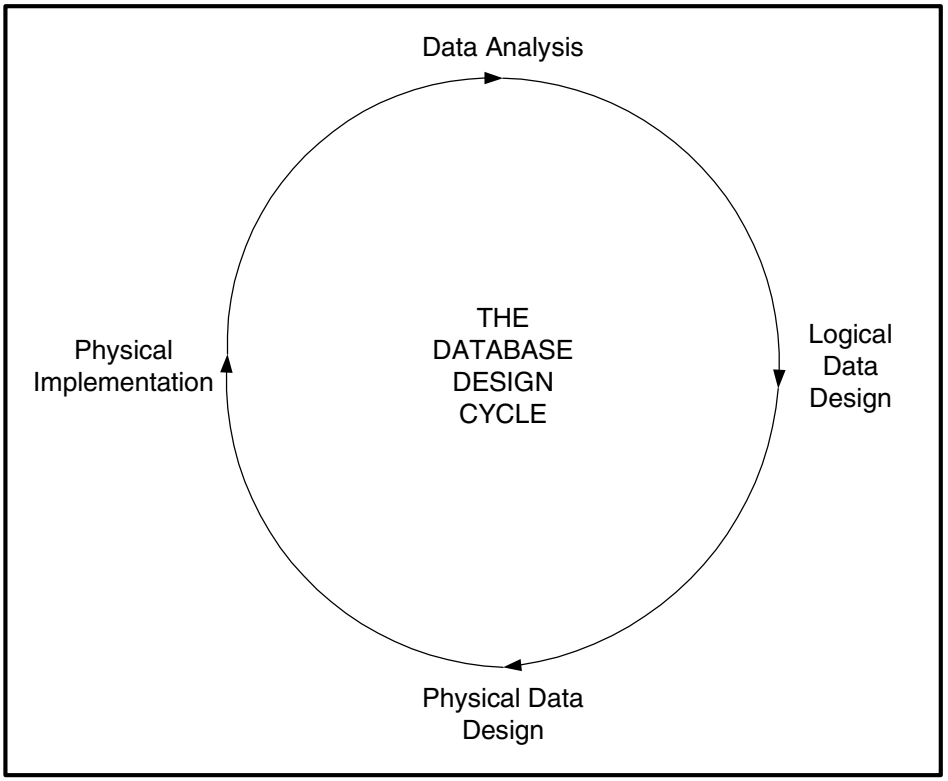


Figure 3–1: Database Design Cycle

You can follow this cycle as a formal or an informal process. There are many tools and methodologies available for following this process; however, it requires research to determine which methodology or tools work best for you.

3.2 Data Analysis

The first step in the database design cycle is to define the data requirements for your business. Answering these questions can help get you started:

- What types of information does my business currently use? What types of information does my business need?
- What kind of information do I want from this system? What kind of reports do I want to generate?

- What will I do with this information?
- What kind of data control and security does this system require?
- Where is expansion most likely?

To answer some of these questions, list all the data you intend to input and modify in your database, as well as all the expected outputs. For example, some of the requirements a retail store might include are the ability to:

- Input data for customers, orders, and inventory items
- Add, update, and delete rows
- Sort all customers' addresses by zip code
- List alphabetically all the customers with outstanding balances of over $1,000
- List the total year-to-date sales and unpaid balances of all the customers in a specific region
- List all orders for a specific item (for example, ski boots)
- List all items in inventory that have less than 200 units, and automatically generate a reorder report
- List the amount of overhead for each item in inventory
- Track customer information so you that have a current listing of customer accounts and balances
- Track customer orders. Print customer orders and billing information for customers and the accounting department
- Track inventory so you know which materials are in stock, which you need to order, where they are kept, and how much of your assets are tied up with inventory
- Track customers' returns on items so you know which items to discontinue or which suppliers to notify

Tables are generally grouped into three types — kernels, associations, and characteristics.

- *Kernels* are tables that are independent entities. They often represent or model things that exist in the real world. Some examples are: customers, vendors, employees, parts, goods, and equipment.
- *Associations* are talbes that represent a relationship among entities. For examples, an order represents an association between a customer and goods.
- *Characteritics* are tables whose purpose is to qulify or describe some other entity. These tables have no meaning in and of themselves, only in relation to the entity they describe. For example, order-lines might describe orders; without an order, an order-line is useless.

The process of identifying the goals of the business, interviewing, and gathering information from the different sources who are going to use the database is a time-consuming but essential process. With the information in hand, you are now ready to define your tables and columns.

3.3 Logical Database Design

Logical database design helps you define and communicate your business' information requirements. When you create a logical database design you describe each piece of information that you need to track and the relationships among, or the business rules that govern, those pieces of information.

Once you create a logical database design, you can verify with users and management that the design is complete (that is, it contains all of the data that must be tracked) and it is accurate (that is, it reflects the correct table relationships and enforces the business rules).

Creating a logical data design is an information-gathering, iterative process. It includes these steps:

- Define the tables you need based on the information your business requires.
- Determine the relationships between the tables.
- Determine the contents (or columns) of each table.
- Normalize the tables to at least the third normal form.
- Determine the primary keys and the column domain. A *domain* is the set of valid values for each column. For example, the domain for the customer number may include all positive numbers.

At this stage, you do not consider processing requirements, performance, or hardware constraints.

3.4 Physical Database Design

The *physical database design* is a refinement of the logical design. In this phase, you examine how the user will access the database. During this phase you determine:

- What data will I commonly use?
- Which column(s) in the table should I index based on data access?
- Where must I build in flexibility and allow for growth?
- Should I denormalize the database to improve performance?

It is possible that at this stage you denormalize the database to meet performance requirements. *Denormalizing* a database means that you re-introduce redundancy into your database in order to meet processing requirements.

Let's look at an example of when you might consider denormalizing a database. In Chapter 2, "Table Relationships and Normalization," you looked at the Order table, which is shown here again in Table 3–1.

Table 3–1: Order Table with Derived Column

Order Number (Primary Key)	Order Date	Order Amount	Total After Tax	Cust Num (Foreign Key)
M31	3/19/91	$400.87	$441.74	101
M98	8/13/91	$3,000.90	$3,300.99	101
M129	2/9/91	$919.45	$1,011.39	101
M56	5/14/90	$1,000.50	$1,100.55	102
M37	12/25/90	$299.89	$329.87	103
M140	3/15/91	$299.89	$329.87	103
M41	4/2/90	$2,300.56	$2,530.61	104

To reduce the table to third normal form, you eliminated the Total After Tax column because it contains data that can be derived. However, now you look at data access requirements. Although you can construct the Total After Tax value, your customer service representatives need this information immediately. Since this is an item of information that is constantly used, you don't want to have to calculate it each time you need it. If you keep it in the database, it is available on request. In this instance, the performance outweighs other considerations.

3.5 Physical Implementation

Once you determine the physical design of your database, you must then consider how the hardware and software components of your development and deployment environments, such as storage devices, disk drive access, backup devices, operating systems, and communications systems can be configured to maximize application performance. See the *Progress Database Administration Guide and Reference* for information on database configuration options.

4

Defining Indexes

This chapter explains what indexes are and when and why to define them.

4.1 Overview

Like a book index, which helps a reader retrieve information on a topic quickly, a database table index speeds up the process of searching and sorting rows. Although it is possible to search and sort data without using indexes, indexes generally speed up data access. Use them to avoid or limit row scanning operations and to avoid sorting operations. If you frequently search and sort row data by particular columns, you might want to create indexes on those columns. Or, if you regularly join tables to retrieve data, consider creating indexes on the common columns.

On the other hand, indexes consume disk space and add to the processing overhead of many data operations—add, update, delete, copy, and move—including data entry, backup, and other common administration tasks. Each time you update an indexed column, Progress updates the index, and related indexes as well. When you create or delete a row, Progress updates each index on the affected tables.

As you move into the details of index design, keep in mind that index design is not a once-only operation. It is a process, and it is intricately related to your coding practices. Faulty code can thwart any index scheme, and masterfully coded queries perform poorly if not properly supported by indexes. Therefore, as the code develops, the indexing scheme—and other aspects of your database design—may have to evolve as well.

Fortunately, index creation, modification, and deletion are all simple operations. Typically, you create a database structure and a set of indexes that support expected access patterns. As the application code develops, new access patterns might arise that require index support, or particular queries might use indexes in unexpected ways, so you might have to modify indexes, query code, or both. Intelligent query coding is outside the scope of this manual, but its relevance to index design and creation cannot be overstated.

4.2 Indexing Databases

This section explains how indexes work and describes their advantages and disadvantages.

4.2.1 How Indexes Work

A database index works like a book index. To look up a topic, you scan the book index, locate the topic, and turn to the pages where the information resides. The index itself does not contain the topic information; instead it contains page numbers that direct you to the appropriate pages. Without an index, you would have to search the entire book, scanning each page sequentially.

Similarly, if you ask for specific data from a database, the system looks for an index. An index contains two pieces of information—the *index key* and and a *row pointer* that points to the corresponding row in the main table. Figure 4–1 illustrates this.

The index table entries are always sorted in numerical, alphabetical, or chronological order. Using the pointers, the system can then access data rows directly and in the sort order specified by the index.

Every table should have at least one index, the *primary index*. When you create the first index on any table in a 4GL implementation, Progress assumes it is the primary index and sets the Primary flag accordingly. Figure 4–1 illustrates the Order-Num (the primary index) and Order-Date indexes on the Order table in the sports database.

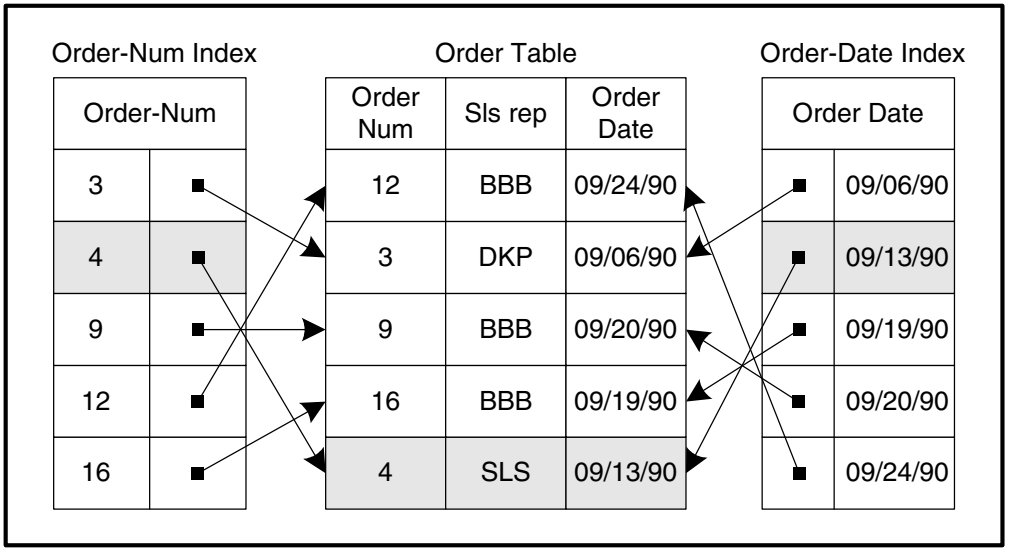


Figure 4–1: Indexing the Order Table

4.2.2 Why Define an Index?

There are four significant benefits to defining an index for a table:

- Direct access and rapid retrieval of rows.

  In Figure 4–1, the rows of the Order table are physically stored in the sequence the user enters them into the database. If you want to find a particular order number or order date, the system must scan every individual row in the entire table until it locates the row(s) that meet your selection criteria. Scanning is inefficient and time consuming where there are hundred thousands or millions of rows in the table.

  However, if you create an index on the Order Number column, Progress then stores the order number values in sorted order (that is, 1, 2, 3, etc.).

  For example, when you query for order number 4, Progress does not go to the main table. Instead, it goes **directly** to the Order-Num index to search for this value. Progress uses the pointer to read the corresponding row in the Order table. Because the index is stored in numerical order, the search and retrieval of rows is very fast.

  Similarly, having an index on the date column allows the system to go directly to the date value that you query (for example, 9/13/90). The system then uses the pointer to read the row with that date in the Order table. Again, because the date index is stored in chronological order, the search and retrieval of rows is very fast.

- Automatic ordering of rows.

  An index imposes an order on rows. Since an index automatically sorts rows sequentially (instead of the order in which the rows are created and stored on the disk), you can get very fast responses for *range* queries. For example, when you query, “Find all orders with dates from 09/6/90 to 09/20/90", all the order rows for that range appear in chronological order.

NOTE: Although an index imposes order on rows, the data stored on disk is in the order in which it was created. So, you can have multiple indexes on a table each providing a different sort ordering, the physical storage order is not controlled by the indexes.

- Enforced uniqueness.

  When you define a unique index for a table, the system ensures that no two rows can have the same value for that index. For example, if order-num 4 already exists and you attempt to create an order with order-num 4, you get an error message informing you that 4 already exists. The message appears because order-num is a unique index for the order table.

- Rapid processing of inter-table relationships.

Two tables are related if you define a column (or columns) in one table that you use to access a row in another table. If the table you access has an index based on the corresponding column, then the row access is much more efficient. The column you use to relate two tables need not have the same name in both tables.

4.2.3 Indexes Used in the Sports Database

Table 4–1 lists some indexes defined in the sports database, showing why the index was defined.

Table 4–1: Reasons for Defining Some Sports Database Indexes *(1 of 3)*

Table	Index Name	Index Column(s)	Primary	Unique
Customer	cust-num	cust-num	YES	YES
–	**Why the Index Was Defined:** 1. Rapid access to a customer given a customer's number. 2. Reporting customers in order by number. 3. Ensuring that there is only one customer row for each customer number (uniqueness). 4. Rapid access to a customer from an order, using the customer number in the order row.			
	name	name	NO	NO
	Why the Index Was Defined: 1. Rapid access to a customer given a customer's name. 2. Reporting customers in order by name.			
	zip	zip	NO	NO
	Why the Index Was Defined: 1. Rapid access to all customers with a given zip code or in a zip code range. 2. Reporting customers in order by zip code.			

Table 4–1: Reasons for Defining Some Sports Database Indexes *(2 of 3)*

Table	Index Name	Index Column(s)	Primary	Unique
Item	item-num	item-num	YES	YES
–	**Why the Index Was Defined:** 1. Rapid access to a item given an item number. 2. Reporting items in order by number. 3. Ensuring that there is only one item row for each item number (uniqueness). 4. Rapid access to an item from an order-line, using the item-num column in the order-line row.			
Order-line	order-line	order-num line-num	YES	YES
–	**Why the Index Was Defined:** 1. Ensuring that there is only one order-line row with a given order number and line number. The index is based on both columns together since neither alone need to be unique. 2. Rapid access to all of the order-lines for an order, ordered by line number.			
	item-num	item-num	NO	NO
	Why the Index Was Defined: 1. Rapid access to all the order-lines for a given item.			

Table 4–1: Reasons for Defining Some Sports Database Indexes *(3 of 3)*

Table	Index Name	Index Column(s)	Primary	Unique
Order	order-num	order-num	YES	YES
–	**Why the Index Was Defined:** 1. Rapid access to a order given an order number. 2. Reporting orders in order by number. 3. Ensuring that there is only one order row for each order number (uniqueness). 4. Rapid access to an order from an order-line, using the order-num column in the order-line row.			
	cust-order	cust-num order-num	NO	YES
	Why the Index Was Defined: 1. Rapid access to all the orders placed by a customer. Without this index, all of the records in the order file would be examined to find those having a particular value in the cust-num column. 2. Ensuring that there is only one row for each suterom/order combination (uniqueness). 3. Rapid access ot the order numbers of a customer's orders.			
	order-date	order-date	NO	NO
	Why the Index Was Defined: 1. Rapid access to all the orders placed on a given date or in a range of dates.			

4.2.4 Disadvantages of Defining an Index

Even though indexes are beneficial, there are two things to keep in mind when defining indexes for your database:

- Indexes take up disk space. (See the "Calculating Index Size" section.)
- Indexes can slow down other processes. When the user updates an indexed column, Progress updates all related indexes as well. Also, when the user creates or deletes a row, Progress changes all the indexes for that table.

Define the indexes that your application requires, but avoid indexes that provide minor benefit or are infrequently used. For example, unless you display data in a particular order frequently (such as by zip code), then you are better off sorting the data when you display it instead of defining an index to do automatic sorting.

4.3 Choosing Which Tables and Columns to Index

If you perform frequent adds, deletes, and updates against a small table, you might not want to index it because of slower performance caused by the index overhead. However, if you mostly perform retrievals, then it is useful to create an index for the table. You can index on the columns that are retrieved most often, and in the order they are most often retrieved.

You don't have to create an index if you are retrieving a large percentage of the rows in your database (for example, 19,000 out of 20,000 rows). It is more efficient to scan the table. However, it is worth your while to create an index to retrieve a very small number of rows (for example, 100 out of 20,000). This way Progress scans only the index table instead of the entire table.

4.4 Indexes and ROWIDs

An index is a list of index values and row IDs (ROWIDs). *ROWIDs* are physical pointers to the database tables that give you the fastest access to rows. ROWIDs do not change during the life of a row—they only change when you dump and reload a database. Also, if you delete a row and create a new identical row, it may have a different ROWID.

A database consists of database blocks. Each block can contain 1024, 2048, 4096, or 8192 bytes of data. Blocks are used to store database rows and indexes. Multiple index entries are stored in a database block. The database blocks of an index are organized into a tree structure for fast access. Progress locates rows within an index by traversing the index tree. Once a row is located, the ROWID accesses the data. Progress does not lock the whole index tree while it looks for the row. It only locks the block that contains the row. Therefore, other users can simultaneously access rows in the same database.

4.5 Calculating Index Size

You can estimate the approximate maximum amount of disk space occupied by an index by using this formula:

Number of rows * (6 + number of columns in index + index column storage) * 2

For example, if you have an index on a character column with an average of 21 characters for column index storage (see Table 4–2) and there are 500 rows in the table, the index size is:

500 * (6 + 1 + 21) * 2 = 29,000 bytes

The size of an index is dependent on four things:

- The number of entries or rows
- The number of columns in the key
- The size of the column values, i.e. the character value “abcdefghi” takes more space than “xyz”
- The number of similar key values

However, you will never reach this maximum because Progress uses a data compression algorithm to reduce the amount of disk space an index uses. In fact, an index uses on average about 20% to 60% less disk space than the maximum amount you calculated using the previously described formula.

The amount of data compressed depends on the data itself. Progress compresses identical leading data as well as collapses trailing entries into one entry. Typically non-unique indexes get better compression than unique indexes.

NOTE: All key values are compressed in the index, eliminating as many redundant bytes as possible.

Table 4–2 lists the column storage values for different data types.

Table 4–2: Column Storage *(1 of 2)*

Data Type	Value	Column Storage in Bytes
Character	1 + *number of characters*, excluding trailing blanks. If the number of characters is greater than 240, add 3 to the number of characters instead of 1.	
Date	3	

Table 4–2: Column Storage *(2 of 2)*

Data Type	Value	Column Storage in Bytes
Decimal	zero	1
	nonzero	2 + (*number of significant digits* + 1)/2
Integer	zero	1
	1	2
	128	3
	32768	4
	8 million	4
	2,147,483,647	5
Logical	false	0
	true	1

Figure 4–2 shows how Progress compresses data.

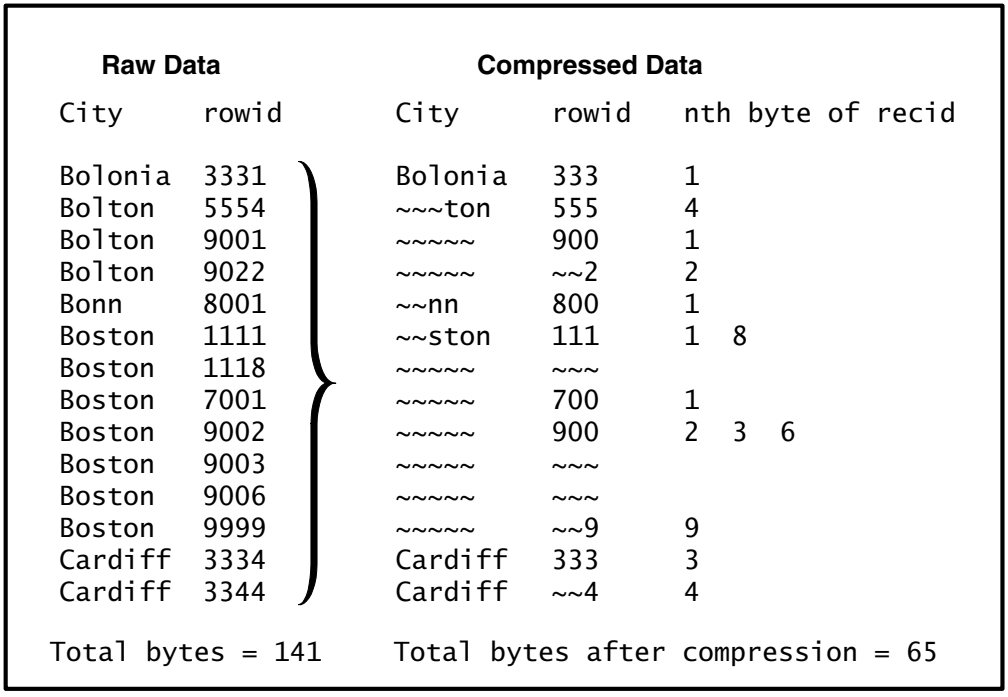


Figure 4–2: Data Compression

The City index is stored by city and by ROWID in ascending order. There is no compression for the very first entry “Bolonia”. For subsequent entries, Progress eliminates any characters that are identical to the leading characters of Bolonia. Therefore, for the second entry, “Bolton”, there is no need to save the first three characters “Bol” since they are identical to leading characters of Bolonia. Instead, Bolton compresses to “ton”. Subsequently, Progress does not save redundant occurrences Bolton. Similarly, the first two characters of “Bonn” and “Boston” (“Bo”) are not saved.

For ROWIDs, Progress eliminates identical leading digits. It saves the last digit of the ROWID separately and combines ROWIDs that differ only by the last digit into one entry. For example, Progress saves the leading three digits of the first ROWID 333 under ROWID, and saves the last digit under nth byte. Go down the list and notice that the first occurrence of Boston has a ROWID of 1111, the second has a ROWID of 1118. Since the leading three digits (111) of the second ROWID are identical to the first one, they are not saved; only the last digit (8) appears in the index.

Because of the compression feature, Progress can substantially decrease the amount of space indexes normally use. In the above example, 65 bytes are used to store the index that previously took up 141 bytes. That's a saving of approximately 54%. As you can see, the amount of disk space saved depends on the data itself. You can save the most space on the non-unique indexes.

4.6 Eliminating Redundant Indexes

If two indexes share the same leading, contiguous components for the same table, they are redundant. Redundant indexes occupy space and slow down performance.

4.7 Deactivating Indexes

Indexes that you seldom use can impair performance by causing unnecessary overhead. If you don't want to delete a seldom-used index, then deactivate it. Deactivating an index eliminates the processing overhead associated with the index, but does not free up its disk space. For information on how to deactivate indexes refer to the *Progress Database Administration Guide and Reference*. To learn how to do this using SQL-92, see the *Progress SQL-92 Guide and Reference*.

NOTE: You would deactivate an index in order to do a bulk load. You would reactivate an index to perform an index rebuild.

5

Progress 4GL Index Usage

This chapter explains how a Progress 4GL application uses indexes.

5.1 Finding Out Which Indexes Are Used

To find out which index Progress uses for a particular query, use the XREF option in the COMPILE statement. The SEARCH lines in the XREF output file show the indexes that are accessed for every record selection statement.

Table 5–1 is a list of tags the XREF compile option generates:

Table 5–1: XREF tags

Tag	Meaning
SEARCH	Indicates an index bracket or look up will be used. The logical database name, table name, and index names are listed. When multiple brackets and indexes are used for the same query, you will see one search line for each bracket.
SEARCH . . . WHOLE-INDEX	Indecates that a suitable bracket could not be constructed and an index scan over the entire table will be performed using the index noted.
SORT-ACCESS	Indicates that the query result is to be ordered by a particular column value and no suitable index exists. A sort of the query result on the noted column value is required.
ACCESS	Indicates that the specified table and field value is used at this point in the program.
CREATE	Indicates that a record is created at this location in the program.
DELETE	Indicates that a record is deleted at this location in the program.
UPDATE	Indicates that the specified field value of a table is updated at this location in the program.

An alternative method to determine index usage is to use the index statistics virtual system table. The startup parameters that enable this are described in Chapter 4 of the *Progress Database Administration Guide and Reference*.

5.2 Maintaining Indexes Through the 4GL

As you work with your application, you will want to know when Progress creates and updates indexes. Progress **creates** a new index entry for a record at the first occurrence of any one of the following:

- At the end of a statement in which Progress assigns values to all components of the index entry
- At the end of the closest iterating subtransaction block in which Progress 4GL creates the record
- When Progress processes a VALIDATE statement
- When Progress releases the record from the record buffer
- At the end of the transaction in which Progress creates the record

Progress **updates** an index at the end of any statement in which it changes the values for one or more index fields. Because Progress updates indexes immediately (at the end of an UPDATE statement), Progress immediately FINDs records in the order of the new index, while the data in the found record is unchanged. Progress changes the data in the record at the end of the scope of the record or when it releases the record.

NOTE: Progress does not update an index if the value you try to assign to the index field is the same as the current value of the index field.

You can change the name of an index at any time. You can also delete nonprimary indexes. However, before letting you delete a primary index, Progress requires that you first designate another index as primary.

If there is only one index, you must create a new index before deleting the existing index. You cannot change any of the component definitions of an index. Instead, you must delete the index and recreate it using the modified component definitions.

Remember that Progress assumes that the first index you create is the primary index, so create your primary index first.

5.3 Using the 4GL ASSIGN Statement

When you want to make changes to several indexed components in a 4GL procedure, use the ASSIGN statement to define these new values. This method allows you to change several values with minimum I/O processing. Otherwise, Progress re-indexes records at the end of each statement that changes the value of an index component.

The following code demonstrates how you can change two index values with the ASSIGN statement:

r-sgn2.p

```
DEFINE VARIABLE neword LIKE order-line.order-num
    LABEL "New Order".
DEFINE VARIABLE newordli LIKE order-line.line-num
    LABEL "New Order Line".
REPEAT:
  PROMPT-FOR order-line.order-num line-num.
  FIND order-line USING order-line.order-num AND line-num.
  SET neword newordli.
  FIND order WHERE order.order-num = neword.
 ASSIGN order-line.order-num = neword
        order-line.line-num = newordli.
END.
```

This procedure changes the order number and line number of an order-line record. (It copies an order-line from one order to another.) It sets the new values into variables and modifies the record with a single ASSIGN statement that contains two assignment phrases in the form *field=expression*. So both fields are changed within a single statement. Because order-num and line-num are used jointly in one index, this method avoids having the indexing done until both values change.

5.4 Indexes and Unknown Values

If an index contains an unknown value (?), Progress sorts that value higher than any other value. When you define a unique index, Progress ensures its uniqueness. For example, if cust-num is a unique index, and there is already a cust-num with a value 10, Progress does not allow you to create a cust-num with the value 10. However,Progress does not prohibit users from entering any number of records with unknown values in index fields. You can prevent users from doing this by making the unique index fields mandatory.

EXAMPLES

Example 1: using the sports database, the following query will display all records where cust-num is > 10 because cust-num is an indexed field and the ? value will sort high in an indexed field:

```
FOR EACH cust WHERE cust-num >10 AND cust-num <= ?
```

However, the query below will display ZERO records because cust-num is the chosen index for the query. Since zip is not the chosen index, the ? value will not sort high and the second part of the query will be false. No records are returned when one part of an AND is FALSE:

```
FOR EACH cust WHERE cust-num >10 AND cust-num <= ? AND zip >0 AND zip <?
```

Example 2: the same rule can affect queries where the ? value is not explicitly used. Using the sports database, if you create three order records where order.cust-num = 1 and order-date = ?, then the following query will return the three records:

```
FOR EACH order WHERE order-date >= 1/1/97
```

However, the following query will return NO records:

```
FOR EACH order WHERE order-date >= 1/1/97 AND cust-num = 1
```

5.5 Indexes and Case Sensitivity

The values for indexed fields are normally stored as all uppercase letters. This ensures that Progress sorts character values properly without regard to case. For example, it treats the character values “JOHN”, “John”, and “john” the same. Also, if “JOHN” is already in a unique index, then any attempt to insert “John” is rejected.

For case-sensitive fields, Progress stores the values exactly as entered. This means that in the above example, it accepts “John” as a different value. Also, when sorted, the uppercase values appear first, then lowercase. So following the same example, “JOHN”, “John”, and “john” all appear in a different order. Note, however, that word indexes on case-sensitive fields are treated as if the field were case insensitive.

Case sensitivity is a characteristic of the field, not the index. Therefore, if an index contains some fields that are case sensitive and some that are not, then the different sorting rules apply.

Field names are not case sensitive; they can be uppercase, lowercase, or a combination of both. If you name a field “Phone” in the Dictionary, you can refer to it as “phone” or “PHONE” in your procedures.

Ordinarily, Progress character fields are not case sensitive (“SMITH”=“Smith”=“smith”). However, on rare occasions, you might want to define a field that is case sensitive. For example, part numbers that contain both uppercase and lowercase characters should be stored in a case-sensitive field. Case-sensitive fields are not recommended, because they depart from standard Progress usage. However, if you require strict adherence to the ANSI SQL standard, you might have to define all character fields as case sensitive. Once a field is defined as case sensitive, you can change it back and forth, unless it is a component of an index. If a field is a component of an index, you must delete the index, then re-create it using the modified field.

Case-sensitive fields can be indexed, and they can be grouped with case-insensitive field components in an index. With case-sensitive indexes, “JOHN”, “John”, and “john” are all unique values. However, sort order depends on the code page of your database. Note that you can (and should) define case-sensitive variables to hold values moving to and from case-sensitive fields. For more information on case sensitivity, see the ANSI SQL (-Q) startup parameter in the *Progress Startup Command and Parameter Reference*.

5.6 How Progress Chooses and Brackets Indexes to Satisfy Queries

Knowing the importance of creating indexes to support common data access patterns is a big step toward efficient design. However, to make your query code and indexes work together effectively, you must understand how Progress chooses indexes to satisfy a particular query.

Efficient query coding is outside the scope of this book, but its relevance to index design and creation cannot be overstated. Therefore, this section briefly explains how Progress chooses the most efficient index or indexes based on the 4GL query. Then, you will learn how Progress brackets an index, when possible, to minimize the number of fetched records.

5.6.1 Background and Terminology

This section provides a concise, abbreviated summary of the concepts and terminology required to discuss how Progress chooses the most efficient indexes to satisfy a query. For the purposes of index selection, there are three general types of WHERE clause:

SYNTAX

```
WHERE searchExpr [ BY field ]
```

For example:

```
WHERE Cust-num > 6
WHERE Name "D"  BY Sales-Rep
WHERE Country = "Zimbabwe"
WHERE Comments CONTAINS "Com*"
```

SYNTAX

```
WHERE searchExpr AND searchExpr [ BY field ]
```

For example:

```
WHERE Cust-num > 6 AND comments CONTAINS "ASAP"
WHERE Name = "Rogers" AND Postal-Code BEGINS "017"
```

SYNTAX

```
WHERE searchExpr OR searchExpr [ BY field ]
```

For example:

```
WHERE Cust-num > 1000 OR Sales-Rep BEGINS "S"
WHERE Postal-Code =< "01500" OR Postal-Code >= "25000"
```

The optional BY *field* clause imposes a sort order on returned records and is called a *sort match*. A *searchExpr* typically has one of the following forms:

BY *field*	**Sort Match**
field = *expression*	**equality match**
field < / =< / > / >= *expression*	**range match**
field BEGINS *expression*	**range match**
wordIndexedfield CONTAINS *stringExpression*	**equality** (simple string) **range** (wildcard string) **none** (>1 string, joined logically)

For more information, see the Record Phrase and FOR statement reference entries in the *Progress Language Reference*.

Because these expressions effectively select the records to return—and the indexes to use—they are called *search conditions*. Commonly, but not always, *field* is an indexed field. Also, a *searchExpr* can include other *searchExpr's* joined by ANDs and ORs, forming arbitrarily complex queries.

The Compiler constructs a logical tree from a query and evaluates both sides of each AND or OR, looking for index criteria. Progress counts equality, range, and sort matches (for OR) and uses them to select and bracket indexes. The precise rules are numerous and complex, and it is not important to fully understand their details. The next sections outline the rules in sufficient detail to help you develop a feel for index usage. In addition, you should experiment by coding various queries, compiling them with the XREF option, and examining index usage as reported in the SEARCH lines of the XREF output file.

The index selection examples that follow are based on the sports database.

5.6.2 Case 1: WHERE *searchExpr*

If there is an index on the *field* in *searchExpr*, or if *field* is the first component in a multi-field index, Progress uses the index. Otherwise, Progress uses the primary index:

Sample WHERE Clause	Indexes Used
`WHERE Customer.Name BEGINS "B"`	Name
`WHERE Customer.Postal-Code BEGINS "01"`	Cust-Num (primary)

If the *searchExpr* references a word-indexed field, Progress uses the word index.

If there is a BY *field* clause, and *field* is indexed, Progress uses the index to sort returned records as long as there is no index on the WHERE clause. If *field* is not indexed, Progress creates a temporary sort table and sorts the records at run time.

5.6.3 Case 2: WHERE *searchExpr* AND *searchExpr*

For a compound WHERE clause, Progress builds a logic tree and evaluates index usage on either side of the AND. When used with the FOR EACH statement, if both sides of the AND include equality matches on all components of non-unique indexes, both indexes are used. When used with the FIND statement, if both sides of the AND are equality matches on indexed fields, only a single index is used. Note that a word index expression with a simple string is an equality match; a wildcard string constitutes a range match:

Sample WHERE Clause	Indexes Used
`WHERE Customer.Name = "Mary"` `AND Customer.Sales-Rep = "Higgins"`	Name Sales-Rep

`WHERE Comments CONTAINS "small"`
`AND Country = "USA"`
`AND Postal-Code = "01730"`
	Comments
	Country-Post

If the selection criteria do not support multiple index usage, see the "General Rules for Choosing a Single Index" section:

NOTE: If Progress uses multiple indexes to select and return records, the precise return order is not predictable. If necessary, you can use the USE-INDEX or BY options to guarantee record return order. In the following example, the BY clause guarantees records are sorted by Cust-Num.

Sample WHERE Clause	Indexes Used

`WHERE Customer.Country = "USA"`
`AND Customer.Sales-Rep = "Higgins"`
`BY Cust-Num`
	Sales-Rep

5.6.4 Case 3: WHERE *searchExpr* OR *searchExpr*

For a compound WHERE clause, Progress builds a logic tree and evaluates index usage on either side of the OR. In general, if all selection criteria on both sides of the OR include matches—equality, range, or sort—on successive, leading components of two non-unique indexes, Progress uses both indexes:

Sample WHERE Clause	Indexes Used
`WHERE Customer.Comments CONTAINS "to*"` `  OR Customer.Name = "Carlin"`	Comments Name
`WHERE Name > "Beaudette"` `  OR Country > "Zambia"`	Name Country-Post

In addition, if one side of the OR includes a CONTAINS clause (that is, it uses a word index), Progress uses the word index and then a second index to satisfy the other side of the OR:

`WHERE Comments CONTAINS "credit"` `  OR Postal-Code > "01000"`	Comments Cust-Num

In this example, the right side of the OR includes a range match, but Postal-Code is the second component of the County-Post index, so the match is not active. Progress uses the primary index to satisfy this piece of the query and, as always, uses the word index to satisfy a CONTAINS clause as shown in this example:

`WHERE Comments CONTAINS "credit"`
`  OR Postal-Code < "01000"`
`  BY Sales-Rep`
	Comments
	Sales-Rep

If the selection criteria do not support multiple index usage, see the "General Rules for Choosing a Single Index" section.

NOTE: If any expression on either side of the OR does not use an index or all its' components, Progress must scan all records using the primary index.

5.6.5 General Rules for Choosing a Single Index

When the selection criteria do not support multiple index usage, Progress uses these general rules (in this order) to select the most efficient index:

1. If there is a CONTAINS clause (which is legal only for word indexed fields), use the word index:

Sample WHERE Clause	Indexes Used
`WHERE Customer.Comments CONTAINS "big"` `AND Customer.Country = "Canada"`	Comments

2. If an index is unique, and all of its components are used in active equality matches, use the unique index. It invariably returns 0 or 1 records:

Sample WHERE Clause	Indexes Used
`WHERE Customer.Cust-Num = 10` `AND Customer.Sales-Rep = "DR"`	Cust-Num

3. Use the index with the most active equality matches. Equality matches are active if:

   - They apply to successive, leading index components

   AND

   - They are joined by ANDs (not ORs or NOTs)

   This disqualifies equality matches on, for example, components 2 and 3 of an index with three components, and it disqualifies matches on components 1 and 2, if they surround an OR:

Sample WHERE Clause	Indexes Used

`WHERE Customer.Country = "Costa Rica"`
`AND Customer.Postal-Code > "3001"`
`AND Customer.Sales-Rep BEGINS "S"`
	Country-Post

Sample WHERE Clause	Indexes Used
`WHERE Customer.Name = "Harrison"` `AND Customer.Sales-Rep BEGINS "S"`	Name

`WHERE Customer.Name = "Harrison"`
`AND (Customer.Country = "Finland"`
`OR Customer.Country = "Denmark")`
	Name

4. Use the index with the most active range matches. For a range match to be active it must stand alone or be connected to other selection criteria by ANDs. In addition, it must apply to an index component having any one of four properties:

   - The component is the first or only one in the index
   - All preceding components in the index key have active equality matches

Sample WHERE Clause	Indexes Used

`WHERE Customer.Sales-Rep = "ALH"`
`AND Customer.Country = "Italy"`
`AND Customer.Postal-Code BEGINS "2"`
	Country-Post

`WHERE Customer.Contact = "DLC"` `AND Customer.Sales-Rep BEGINS "S"`	Sales-Rep
`WHERE Customer.Contact = "Ritter"` `AND Comments CONTAINS "compute*"`	Comments

5. Use the index with the most sort matches. (All sort matches are active.)

Sample WHERE Clause	Indexes Used

`WHERE Customer.Country BEGINS "EC"`
`AND Customer.Sales-Rep BEGINS "S"`
`BY Country`
	Country-Post

`WHERE Customer.Contact = "Wilson"`
`AND Customer.Credit-Limit > 2000`
`BY Name`
	Name

`WHERE Name = "Wilson"`
`OR Customer.Credit-Limit = 2000`
`BY Sales-Rep`
	Sales-Rep

6. Use the index that comes first alphabetically. That is, if there is a tie—if multiple indexes have the same number of active equality, range, and/or sort matches—use the alphabet to decide:

Sample WHERE Clause	Indexes Used
`WHERE Customer.Name = "Samali"` `AND Customer.Sales-Rep = "BCW"`	Name
`WHERE Customer.Country BEGINS "EC"` `AND Customer.Sales-Rep BEGINS "B"`	Postal-Code

7. Use the primary index:

Sample WHERE Clause	Indexes Used

`WHERE Customer.Contact = "MK"`
`AND (Customer.Sales-Rep BEGINS "S"`
`OR Customer.Sales-Rep BEGINS "B")`
	Cust-Num

`WHERE Customer.Postal-Code >= "01000"` `AND Customer.City = "Boston"`	Cust-Num
`WHERE "meaningless expression"`	Cust-Num

5.6.6 Bracketing

Having selected one or more indexes to satisfy a query, Progress tries immediately to isolate the smallest necessary index subset, so as to return as few records as possible. This is called *bracketing*. Careful query design can increase the opportunities for bracketing, thereby preventing Progress from scanning entire indexes and examining all records. The rules for bracketing are simple:

- Bracket on active equality matches.
- Bracket an active range match, but no further brackets are possible for that index.

The following table provides some bracketing examples:

Sample WHERE Clause	Indexes Used	Brackets

`WHERE Contact = "DLC"`
`AND (Sales-Rep BEGINS "S"`
`OR Sales-Rep BEGINS "B")`
	Cust-Num	None

`WHERE Postal-Code >= "01000"` `AND City = "Boston"`	Cust-Num	None
`WHERE Name = "Harrison"` `AND Sales-Rep BEGINS "S"`	Name	Name
`WHERE Contact = "DLC"` `AND Sales-Rep BEGINS "S"`	Sales-Rep	Sales-Rep

`WHERE Country BEGINS "EC"`
`AND Sales-Rep BEGINS "S"`
`BY Country`
	Country-Post	Country-Post

`WHERE Comments CONTAINS "big"`
`AND Country = "USA"`
`AND Postal-Code = "01730"`
	Comments
	Country-Post
		Country
		Postal-Code

The following recommendations are intended to help you maximize query performance. They are only recommendations, and you may choose to ignore one or more of them in specific circumstances.

- Avoid joining range matches with AND.
- Avoid ORs if any expression on either side of the OR does not use an index (or all its components), be aware that Progress must scan all records using the primary index.

- With word indexes, avoid using AND with two wildcard strings, either in the same word index (WHERE comments CONTAINS “fast* & grow*”) or in separate word indexes (WHERE comments CONTAINS “fast*” AND report CONTAINS “ris*”).
- Avoid WHERE clauses that OR a word index reference and a non-indexed criterion (WHERE comments CONTAINS “computer” OR address2 = “Bedford”).

5.7 Index-related Hints

- As the ratio of records that satisfy the query to total records decreases, the desirability of indexed access increases.
- A rule-of-thumb is fewer indexes per table with heavy update activity and limited access; more indexes per table with heavy access but limited update activity.
- Most index processing—including, for example, all word index evaluations—takes place on the Progress server side, and the minimum required records are returned to the client.

6

Progress 4GL Word Indexes

Progress supports a type of index called a word index. A *word index* contains all the words from a character field or array of character fields. Searching with this index makes it quick and easy to locate records that contain specific words.

You can define a word index the same way you define any index from within the Data Dictionary. The following sections describe the methods for defining word indexes using the character and graphical Data Dictionary.

6.1 Word Index Support

To support word indexes, the CONTAINS option is included in the WHERE clause of the Record phrase. This allows you to use a word index in a FOR EACH statement. For example, suppose you had defined a word index on the address field of the Customer table. The following statement finds customer records that include the word Homer within the address field:

```
FOR EACH customer WHERE address CONTAINS "Homer":
```

This is the general syntax of the WHERE clause with the CONTAINS option:

SYNTAX

```
WHERE field CONTAINS string-expression
```

field

Refers to a character field or array on which a word index has been defined.

string-expression

Contains one or more words to search for in the character field *field:*

SYNTAX

```
"word [ [ & | | | ! | ^ ] word ] ... "
```

word

A word to search for or a wildcard string. A wildcard string is any word that ends with an asterisk (*). You can use a wildcard string to search for words that begin with a specific initial substring. For example, the string “sales*” matches the words sales, salesman, and salesperson.

& | | | ! | ^

Word separators. The ampersand (&)represents a logical AND; a vertical line (|), exclamation point (!), or caret (^) represents a logical OR. Separators allows you to limit your search to records that contain all words that you specify or to enlarge your search to all records that contain any of the words you specify. You can combine AND and OR operations within a string expression and group items with parentheses to create complex search conditions.

EXAMPLE

The following example assumes that a word index has been defined on the Terms field of the order table:

```
FOR EACH order WHERE Terms
    CONTAINS "free | gratis | (no & charge)":
  DISPLAY order.
END.
```

The AND operator (&) takes precedence over the OR operator (|). For this reason, the parentheses in the preceding example are not required. They are used only to improve readability.

EXAMPLE

This example finds all books whose description contains both the words “computer” and “science”. If you omit both the ampersand and vertical line between two words (that is, the words are separated only by spaces), a logical AND is assumed:

```
FOR EACH book WHERE description CONTAINS "computer science":
    DISPLAY book.
END.
```

If your search condition specifies that the record must contain two or more specific words, the order of the words is insignificant. In the previous example, a record containing the word “science” and later the word “computer” may be displayed.

EXAMPLE

To find words in exact order, you can combine CONTAINS with the MATCHES function. In this example, the words “computer” and “science” must appear in exact order, although another word or words can appear between them:

```
FOR EACH book WHERE description
    CONTAINS "computer science" and description
      MATCHES "*computer *science*":
    DISPLAY book.
END.
```

EXAMPLE

You can combine other search criteria with a CONTAINS option. This example searches a table that contains information about traffic accidents. It finds only records for accidents that

occurred in Boston and contain the word “milk” and either the word “truck” or “trailer” in the description field:

```
FOR EACH accident WHERE city = "Boston" AND
        description CONTAINS "milk (truck ^ trailer)":
```

NOTE: Use the Stash Area (-stsh) startup parameter when you have a word index on a large character field. For more information about this startup parameter, see the description of the Stash Area (-stsh) startup parameter in the *Progress Startup Command and Parameter Reference*.

6.2 Word Delimiters

Progress maintains a table of word characters and word delimiters that it uses to break text into words to be put in the word index. Progress defines a default set of rules. You can establish your own rules to override the Progress defaults.

Progress defines each character to have one of eight attributes:

- **LETTER** — A letter is always part of a word.
- **DIGIT** — A digit is always part of a word.
- **USE_IT** — A character with this attribute is always part of a word.
- **BEFORE_LETTER** — A character with this attribute is treated as part of a word only if followed by a character with the LETTER attribute. In all other cases, the character is not part of a word and is treated as a word separator.
- **BEFORE_DIGIT** — A character with this attribute is treated as part of a word only if followed by a character with the DIGIT attribute.

- **BEFORE_LET_DIG** — A character with this attribute is treated as part of a word only if followed by a character with either the LETTER or DIGIT attribute.
- **IGNORE** — A character with this attribute is ignored when building the word index. Such a character is never part of a key value.
- **TERMINATOR** — A character with this attribute ends a word and is never considered part of a word.

6.2.1 Progress Defaults

The default rules defined by the Progress 4GL are as follows:

- The LETTER attribute is assigned to all characters defined as letters in the current language. In English, the uppercase characters A-Z and the lower-case characters a-z are defined as letters.
- The DIGIT attribute is assigned to the characters 0-9.
- The following characters are assigned the USE_IT attribute: dollar sign ($), percent sign (%), number sign (#), at symbol (@), and underline (_).
- The following characters are assigned the BEFORE_DIGIT attribute: period (.), comma (,), and hyphen (-). This means, for example, “12.34" is one word, but “ab.cd” is two words.
- The IGNORE attribute is assigned to the apostrophe (’). This means, for example, the word John’s is the same as Johns.
- The TERMINATOR attribute is assigned to all other characters.

These rules apply to both the text in a word index field and text in a CONTAINS clause. For example, if you type “John’s” in a CONTAINS clause, it is treated the same as “Johns”. If you type “8:30pm” it is treated as two words: 8 and 30pm. Therefore, a record containing the string “8:30pm” will be found, but so could a record containing “8 Capital St. at 7:30pm.” As with indexes, separators and terminators are ignored for the second part of the CONTAINS clause. This means that the string entered in the database matches the string entered after the CONTAINS clause.

6.2.2 Defining Your Own Delimiters

To define attributes for your own delimiters, you must create a *word-break rules file*. Within this file, you define a data structure named word_attr. The following is an example of a word-break rules file:

```
#define    AT_SIGN    64
#define    PERIOD     46
#define    COMMA      44

word_attr  =
{
PERIOD,      BEFORE_DIGIT,
COMMA,       BEFORE_DIGIT,
0x2D,         BEFORE_DIGIT,      /*hyphen */
39,          IGNORE,            /*single quote */
'$',          USE_IT,
'%',          USE_IT,
'#',          USE_IT,
AT_SIGN,      USE_IT,
'_'          USE_IT
};
```

Note that you can use the #define syntax to define constants. Within the word_attr definition you can reference characters by enclosing them in single quotes (‘ ’), by their decimal ASCII values, or by their hexadecimal values (0x2D, for example). Each item in the table except the last item must be followed by a comma.

Your word-break table does not have to include every character. By default, all letters in the language are assigned the LETTER attribute, the characters 0 through 9 are assigned the DIGIT attribute, and all other characters are assigned the TERMINATOR attribute. You only have to specify the characters whose attributes you want to change.

NOTE: The asterisk (*), vertical line (|), exclamation point (!), caret (^), and opening and closing parentheses all have special meaning within a CONTAINS clause. You **must** give these characters the TERMINATOR attribute or their special meaning is lost. You may, however, select to maintain only one of the OR operators and use the others as letters.

After you’ve defined your word-break table, you must compile it with the PROUTIL command:

Operating System	Syntax
UNIX Windows	`proutil` *`database`* `-C wbreak-compiler` *`src-file rule-numb`*

In the syntax, *database* is the name of your database, *src-file* is the name of your word-break rules file, and *rule-numb* is a number between 1 and 255 that uniquely identifies this set of rules on your system.

The PROUTIL command produces a binary file named prowrd.*rule-numb*. For example, if rule-numb is 34, the file is named prowrd.34. To reference this file, you must either move it to the $DLC directory or set the environment variable PROWD*rule-numb* to reference it. For example, the variable PROWD34 specifies the location of the prowrd.34 file. (Note that the PROWD environment variable does not contain a period.)

To apply word-break rules to a database, use the word-rules qualifier of the PROUTIL command:

Operating System	Syntax
UNIX Windows	`proutil database -C word-rules rule-numb`

The value of *rule-numb* is the same value you specified when compiling the rules. To switch back to the default rules, specify 0 for *rule-numb*.

If you change the word-break rules when word indexes are active, the indexes might not work properly because the rules used to create the index differ from those used when searching the index. Therefore, when you change the break rules for a database, Progress warns you if any word indexes are active. You should rebuild these indexes. You can make old indexes consistent with the new rules by rebuilding them with the PROUTIL idxbuild qualifier. For more information about the PROUTIL idxbuild qualifier, see the PROUTIL idxbuild qualifier entry in the *Progress Database Administration Guide and Reference*.

Progress maintains a cyclic redundancy check (CRC) to ensure that the word-break rule file does not change between sessions. If it has changed, Progress displays a message when you attempt to connect to the database. The connect attempt fails. You can fix this by restoring the original file or resetting the break rules to the default. Note that resetting to the default break rules may invalidate your word indexes.

6.3 Word Indexing External Documents

If you want to create a word index on an existing document or documents, you must first import the text into a Progress database. You may choose to index the text by line or by paragraph.

6.3.1 Indexing by Line

To index a set of documents by line, you might create a table called line with three fields: document_name, line_number, and line_text. Define the primary index on document_name and a word index on line_text. Next, write a text loading Progress program that reads each document and creates a line record for each line in the document. To improve storage efficiency, you might replace the document_name field with a document_number field, and create a document table to associate a document_name with each document_number.

You will have to update your index when the base documents change. One method is to store a document ID as part of the record for each line. When a document changes, you can delete all lines with that document ID and reload the document.

The following code queries the line table using the word index:

```
DEFINE VARIABLE words AS CHAR FORMAT "x(60)"
        LABEL "To find document lines, enter search words".

REPEAT:
    UPDATE words.
    FOR EACH line WHERE line_text CONTAINS words:
      DISPLAY line.
    END.
END.
```

The example prompts for a string of words and then displays each line that matches the search criteria.

6.3.2 Indexing by Paragraph

Instead of indexing each line of text, you might want to index by paragraph. The method to use is similar to that for line indexing, but the text from a paragraph can be much longer. Therefore, instead of defining the field line_text, define an array field paragraph_text. If the longest paragraph is about 600 characters, specify 10 as the extent of paragraph_text.

When reading the document into the database, break each paragraph into units of about 60 characters and assign each to a member of the paragraph_text array. Note that you must break the text at a space or other word delimiter. Do not break the text in the middle of a word, because that causes the two fragments of the word to be indexed as two words.

You can use the index by paragraphs in the same way you use an index by lines. You can also index by chapter, by page, and so forth.

6.4 Word Indexing and Non-Progress Databases

Word indexing is a Progress database feature that is not supported by other databases; however, you can simulate it using this method. To create a word index on text from a non-Progress database, you must write a Progress 4GL routine to read records from the non-Progress database and copy the text into a table in a Progress database.

You might define a table with the fields foreign primary key and foreign text. Define a word index on foreign_text and load text and keys from the non-Progress database into the table. You can then use the word index to find the primary keys for records that contain specific words.

6.5 Word Indexing and SQL-92

Currently, there is no concept of a word index in SQL-92. Therefore, there is no implementation of the CONTAINS operator.

NOTE: SQL-92 clients are able to perform data maintenance on a record that contains a column indexed with a word index. The index information will be handled correctly for use by future 4GL queries.

7

Constraints and Indexes Using SQL-92

This chapter explains what constraints and indexes are and why to define them.

7.1 Constraints

You can achieve data integrity with constraints, or rules, using SQL-92 contructs on your keys. Constraints can be named at the column or table level.

7.1.1 Keys

There are two types of keys: primary and foreign.

A *primary key* is a column (or group of columns) whose value uniquely identifies each row in a table. Because the key value is always unique, you can use it to detect and prevent duplicate rows. You can have only one primary key per table. Primary keys may be referenced by other tables. There can be only one primary key for a table. For more information on criteria for choosing a primary key refer to Chapter 1, "Introduction."

A *foreign key* is a column of data that is common to more than one table (and it is the primary key of one of these tables). It is this relationship of a column in one table to a column in another table that provides the relational database with its ability to join tables. Chapter 2, "Table Relationships and Normalization," describes this concept in more detail.

When using SQL-92, you create your keys when you create your table. For information on the SQL CREATE TABLE command see the *Progress SQL-92 Guide and Reference*.

7.1.2 PRIMARY Constraint

Use this constraint to name one or more columns to uniquely identify a row in a table. There can only be one primary key for a table. These columns cannot contain null values.

7.1.3 UNIQUE Constraint

Use this constraint to name one or more columns to uniquely identify a row in a table. These columns cannot contain null values.

7.1.4 FOREIGN Constraint

This constraint references a primary or unique key in another table. Ensures the value must equal the primary or unique key in the referenced table.

7.1.5 NOT NULL Constraint

Use the NOT NULL constraint to ensure that no rows in a table will have a null value in the specified column.

7.1.6 CHECK Constraint

The CHECK constraint is the most flexible constraint. You specify a search condition that can test for a range of values, a minimum or maximum value or a list of values. The CHECK constraint refers to columns in the specified table only and cannot contain queries or aggregate functions.

7.2 Indexes

An index is a "b-tree" that corresponds with a data table. For every row in the data table there is a corresponding row in the index table. Records in the index are always sorted.

The best time to create an index is the same time you are creating the table. It is important to anticipate the ways in which you may want to access the data and create indexes to accommodate that.

A primary index is an index on a primary key. A secondary index is an index on a key that is not the primary key in the table. An index can be unique.

An index has these advantages:

- Faster row search and retrieval. It is more efficient to locate a row by searching a sorted index table than by searching an unsorted table.
- Records are ordered automatically to support your particular data access patterns. No matter how you change the table, when you browse or print it, the rows appear in indexed order instead of their stored physical order on disk.
- When you define an index as unique, each row is unique. This ensures that duplicate rows do not occur.
- A combination of columns can be indexed together to allow you to sort a table in several different ways at once (for example, sort the Projects table by a combined employee **and** date column).
- Efficient access to data in multiple related tables.

To create an index in SQL-92 use the CREATE INDEX statement. This statement will create an index on the table you specify using the specified columns of the table. For more information on the CREATE INDEX statement see the *Progress SQL-92 Guide and Reference*.

You may create a unique index by creating the index on the primary key. A unique index does not allow the table to contain any rows with duplicate column values for the set of columns specified for the index.

A secondary index may be created on a key that is not primary.

7.2.1 SQL-92 Notes

- The index is built immediately for all existing rows in the table.
- If you specify an index as UNIQUE and all values in the existing rows are not unique, an error message is returned and the index is not created.
- All indexes created by SQL-92 are case-sensitive.

8

Progress 4GL Triggers

A database trigger is a block of 4GL code that executes whenever a specific database event occurs. A database event is an action preformed against the database.

This chapter discusses Progress 4GL triggers.

8.1 Trigger Definition

A 4GL trigger is a block of 4GL code that executes whenever a specific database event occurs. A database event is an action performed against the database. For example when you write a record to a database, a WRITE event occurs.

Because database triggers execute whenever a database event occurs, they are useful for tasks such as referential integrity. For example, if you delete a customer record from a database, you may also want to delete all of the customer's order records. The event (deletion of a customer record) initiates the task (deletion of all associated order records). A database trigger is ideal for this type of processing, because the same task must always be performed when the particular event occurs. Other suitable tasks are maintaining database security or writing database audit trails.

For more information on using the Progress 4GL to write trigger code, see the *Progress Language Reference* and the *Progress Programming Handbook*.

8.2 4GL Database Events

Database triggers associate a table or field with a database event. When the event occurs, the trigger executes. Progress does not provide events for all database actions. For example, although you can dump database definitions from a database, you cannot write a trigger for a DUMP event, because Progress does not provide a DUMP event.

However, Progress does provide replication-related triggers in addition to standard triggers for certain events. Replication-related triggers help you implement database replication. For more information on replication-related triggers, refer to the *Progress Programming Handbook*.

The database events that Progress 4GL supports follow.

8.2.1 CREATE

When Progress executes a CREATE or INSERT statement for a particular database table, Progress creates the record, then fires all applicable CREATE triggers, then fires all applicable REPLICATION-CREATE triggers.

8.2.2 DELETE

When Progress executes a DELETE statement for a particular database table, Progress fires all applicable DELETE triggers, then fires all applicable REPLICATION-DELETE triggers, then validates the delete, then performs the delete.

8.2.3 FIND

When Progress reads a record in a particular database table using a FIND or GET statement or a FOR EACH loop, Progress fires all applicable FIND triggers. FIND triggers fire only for records that completely satisfy the full search condition, such as a WHERE clause specifies. FIND triggers do not fire in response to the CAN-FIND function.

Note that if a FIND trigger fails, Progress behaves as though the record had not met the search criteria. If the FIND is within a FOR EACH block, Progress simply proceeds to the next record. If your application uses the BREAK option of the PRESELECT phrase (which forces Progress to retrieve two records at a time, so it can find the break), Progress executes the FIND trigger twice during the first FIND, which is actually two FINDs in succession. Thereafter, Progress looks one record ahead of the record currently in the record buffer, and executes the FIND trigger before it places the next record in the buffer.

8.2.4 WRITE

When Progress changes the contents of a record and validates it for a particular database table, Progress first fires all applicable WRITE triggers, then fires all applicable REPLICATION-WRITE triggers. Progress automatically validates a record when releasing it. You can also use the VALIDATE statement to explicitly validate a record. In either case, WRITE triggers execute before the validation occurs (so WRITE triggers can correct values and do more sophisticated validation). Progress might execute the WRITE triggers for a single record more than once before it writes the record to the database (if it validates the record more than once and you modify the record between validations).

8.2.5 ASSIGN

When Progress updates a particular field in a database record, Progress fires all applicable ASSIGN triggers. Unlike the other database events, this one monitors a specific field rather than a table. ASSIGN triggers execute when the contents of the associated field are modified. The trigger procedure executes at the end of a statement that assigns a new value to the field and after any necessary re-indexing. If the statement contains several field assignments (for example, UPDATE name city st), Progress fires each applicable ASSIGN trigger at the end of the statement. If any trigger fails, Progress undoes the statement (unless the code specifies NO-UNDO). For more information on replication-related triggers and database replication, see the *Progress Database Administration Guide and Reference*.

8.3 Schema and Session Database Triggers

Progress 4GL supports two types of database triggers: schema and session. A schema trigger is a .p procedure that you add, through the Data Dictionary, to the schema of a database. Schema trigger code defined in the database is executed by database clients.

A session trigger is a section of code that you add to a larger, enclosing procedure.

8.3.1 Schema Triggers

You create schema triggers through the Table or Field Properties dialog box in the Data Dictionary. When you use the Data Dictionary to define a schema trigger for a table or field, the trigger is automatically added to the table or field's data definitions. Progress allows you to define the trigger while you are creating or modifying a table or field. This trigger definition is stored in a trigger procedure. For information on using the Data Dictionary to create and delete triggers, see the *Progress Basic Database Tools* (Character only) and in graphical interfaces, the online help for the Data Dictionary. For more information schema triggers, see the *Progress Programming Handbook*.

8.3.2 Differences Between Schema and Session Triggers

Although their syntax is slightly different, schema and session triggers provide similar functionality. The important difference between them is that schema triggers are independent procedures; whereas session triggers are contained within a larger procedure. Because of this difference, schema triggers always execute when a specified event occurs, regardless of what application initiates the event.

Session triggers are defined as part of a particular application and are only in effect for that particular application. Since session triggers are executed from within an enclosing procedure, they have access to the frames, widgets, and variables defined in the enclosing procedure.

Since schema triggers are compiled separately from the procedure that initiates their execution, they do not have access to the procedure's frames, widgets, and variables. Use schema triggers for processing that you always want to perform for a specific event. For example, when an order record is deleted, you might always want to delete the corresponding order-line records.

Use session triggers to perform additional or independent processing when the event occurs. Both types of trigger scan return ERRORS that cause the associated event to fail. For more information on the ERROR option of the RETURN statement, see the *Progress Language Reference*.

8.3.3 Trigger Interaction

You can define a schema and a session trigger for the same table/event or field/event pair. How the triggers interact depends on how you define them.

Ordinarily, both triggers execute, with the session trigger executing first (except for the FIND session trigger, which executes after the FIND schema trigger). In this way, the schema trigger always has precedence over the session trigger. For a WRITE, DELETE, CREATE, or ASSIGN event, the schema trigger can override the session trigger. For a FIND event, the schema trigger can preempt the session trigger.

8.4 General Considerations

When you write 4GL triggers, there are several general considerations that you should keep in mind.

8.4.1 Metaschema Tables

Progress does not allow any database triggers on events for metaschema tables and fields (tables or fields named with an initial underscore). You can only intercept database events for an application database object.

8.4.2 User-interaction Code

Progress allows you to include any type of 4GL statement within a database trigger block, including those that involve user interaction. However, it is not a recommended to include any statements that call for input from the user.

For example, if the user runs your procedure in batch mode, a trigger with a prompt causes the procedure to stop, waiting for user input. Also, if an application is written in ESQL, it can not receive user input.

NOTE: 4GL triggers that contain user interaction may fail with an error at run-time if they are invoked by an environment (such as ESQL-89) that has no user interface.

8.4.3 FIND NEXT and FIND PREV

You cannot delete or advance any record in a buffer passed to a database trigger procedure (as with a FIND NEXT or FIND PREV statement) within the trigger procedure.

8.4.4 Triggers Execute Other Triggers

An action within one trigger procedure may execute another trigger procedure. For example, if a trigger procedure assigns a value to a field and you defined an ASSIGN trigger for that field,

that ASSIGN trigger executes. You must carefully design your triggers to avoid conflicts. For example, trigger A could change data, which could cause trigger B to execute. Trigger B could change the same data, a change you did not anticipate or want.

8.4.5 Triggers Can Start Transactions

By their nature, CREATE, DELETE, WRITE, and ASSIGN triggers execute from within a transaction. (The triggering event itself must occur within a transaction.)

This means that while you can start a subtransaction within a trigger procedure for these events, you cannot start a new transaction. However, a FIND may or may not be executed within a transaction. Therefore, you should not assume that you can start a subtransaction within a FIND trigger; it might turn out to be a complete transaction.

8.4.6 Where Triggers Execute

Database triggers (including replication-related triggers) execute in the application logic, which consists of Progress AppServers and local 4GL procedures. For more information on Progress AppServers, see *Building Distributed Applications Using the Progress AppServer*.

8.4.7 Storing Trigger Procedures

You can store a database's trigger procedures in an operating system directory or in a Progress 4GL r-code library.

8.4.8 SQL Considerations

Because of inherent differences between the Progress 4GL and SQL, triggers may not execute in exactly the same way. For more information on how SQL and database triggers interact, see the *Progress SQL-92 Guide and Reference*.

NOTES

- SQL-92 triggers will not fire when 4GL events occur.
- 4GL triggers will no fire when SQL events occur.
- To ensure integrity, you must have both types of triggers, or use SQL to only read data, or have only SQL table.

9

Java Stored Procedures and Triggers

A Java stored procedure is a group of SQL-92 statements that form a logical unit and perform a particular task. Stored procedures are used to encapsulate a set of operations or queries for execution on a database server.

A trigger is a special type of stored procedure that is automatically invoked when certain database events occur. Triggers are used to enforce referential integrity and business rules.

This chapter discusses stored procedures and triggers.

9.1 Java Stored Procedures

A Java stored procedure is a group of SQL-92 statements that form a logical unit and perform a particular task. Stored procedures are used to encapsulate a set of operations or queries to execute on a database server.

Stored procedures can be compiled and executed with different parameters and results. They may have any combination of input, output and input/output parameters. Stored procedures do not have to be tied to database events.

Java stored procedures cannot be called from Progress 4GL clients.

9.1.1 Advantages of Stored Procedures

Stored procedures provide a flexible mechanism in which to store a collection of SQL-92 statements and Java program constructs in a database.

Stored procedures can also improve performance.

9.1.2 How Progress SQL-92 Interacts with Java

Progress stored procedures allow the use of standard Java programming constructs along with standard SQL-92 statements.

When you create a stored procedure, the SQL-92 server processes the Java code, submits it to the Java compiler, receives the compiled results and stores the result in the database.

When an application calls a stored procedure, the SQL-92 server interacts with the Java Virtual Machine (JVM) to execute the stored procedure and receive any results.

For more information on creating stored procedures see the *Progress SQL-92 Guide and Reference*.

9.2 SQL-92 Triggers

A trigger is a procedure that is invoked when certain database events occur. Triggers are used to enforce referential integrity and business rules.

A Java trigger is a Java routine that is executed by a SQL-92 server process when certain database events occur (INSERT, UPDATE, and DELETE) on the trigger's target table. Triggers use Progress Java classes.

Triggers can be thought of as automatically called stored procedures. Like stored procedures, triggers are stored in the database.

Designing the trigger code consists of writing the Java code that defines what the application does when the user interacts with the user-interface elements. For example, the application might write a particular database record when the user clicks on a particular button. In this case, the trigger code is the code that writes the record when the user clicks on that button.

Java triggers execute on the database server side.

9.2.1 Typical Uses for Triggers

Cascading Deletes

A delete operation on one table causes additional rows to be deleted from other tables that are related to the first table by key values. This is an active way to enforce referential integrity that a table constraint enforces passively.

Cascading Updates

An update operation on one table causes additional rows to be updated in other tables that are related to the first table by key values. These updates are commonly limited to the key fields themselves. This is an active way to enforce referential integrity that a table constraint enforces passively.

Summation Updates

An update operation in one table causes an update operation in a row of another table. The second value is increased or decreased.

Automatic Archiving

A delete operation on one table creates an identical row in an archive table that is not otherwise used by the database.

9.2.2 Trigger Structure

A trigger has a specification and a body. The body of the trigger has BEGIN and END delimiters enclosing a Java snippet. The Java snippet contains the sequence of Java statements that define the triggered action. This action is executed when the trigger is fired.

For more information on creating Java triggers see the *Progress SQL-92 Guide and Reference*.

9.2.3 Trigger Types

There are two types of Java triggers provided: statement and row.

Statement triggers are tied to the SQL-92 statement boundaries. These triggers are only observed through the SQL-92 interface. Row triggers are fired when database events occur regardless of the interface to the database.

EXAMPLE

The following code illustrates an example of a statement trigger:

```
CREATE TRIGGER trig1 BEFORE DELETE ON customer
FOR EACH STATEMENT
BEGIN
   SQLIStatement upd_stmt
     (“UPDATE summary SET trans_tot = trans_tot + 1");
   upd_stmt.execute ();
END
```

NOTE: The UPDATE statement executes only once regardless of how many rows are deleted.

EXAMPLE

The following code examples illustrates a row trigger:

```
CREATE TRIGGER trig1
BEFORE INSERT ON customer REFERENCING NEWROW
FOR EACH ROW
BEGIN
  String new_name; Integer new_num;
  new_num = (Integer) NEWROW.getValue(1,INTEGER);
  new_name = (String) NEWROW.getValue(2,INTEGER);
  SQLIStatement ins_stmt (“INSERT INTO customer2 (?,?,?,?,?,?) “);
  “UPDATE summary SET trans_tot = trans_tot + 1");
  ins_stmt.setParam (1, new_num);  ...
  ins_stmt.execute ();
END
```

NOTES

- The INSERT statement executes once for each record inserted into the customer table.
- In order for a trigger to work, you cannot use a FOREIGN KEY restriction in the table definition.

9.2.4 Differences between 4GL and Java Triggers

There are differences in how Progress handles and supports Java versus 4GL triggers.

- SQL clients and servers do not execute 4GL-based triggers
- 4GL clients and servers do not execute Java triggers
- 4GL does not observe SQL constraints

Additionally, if you need both 4GL and SQL access to your triggers, you must create both.

Restrictions

The actions in a Java database trigger are restricted. Java triggers:

- cannot contain DDL statements
- cannot perform commit or rollback statements
- use a restricted set of Java libraries
- cannot perform an OS operation
- do not support ASSIGN or FIND triggers

9.3 Triggers versus Stored Procedures

Triggers are identical to stored procedures in many respects. There are three main differences:

- Triggers are automatic.

  When the trigger event (INSERT, UPDATE or DELETE) affects the target table, the Java code contained in the body of the trigger executes. A stored procedure must be explicitly invoked by an application or another procedure.

- Triggers cannot have output parameters or a result set.

  Since triggers are automatic, there is no calling application to process any output they might generate.

- Triggers have limited input parameters.

  The only possible input parameters for triggers are values of columns in the rows affected by the trigger event.

9.4 Triggers versus Constraints

Triggers are ideal for enforcing referential integrity because they are automatic. In this regard, they are similar to constraints since both triggers and constraints can help ensure that a value stored in the foreign key of a table must either be null or be equal to some value in the matching unique or primary key of another table.

Triggers differ from constraints in the following ways:

- Triggers are active.

  Constraints are passive. While constraints prevent updates that violate referential integrity, triggers perform explicit actions in addition to the update operation.

- Triggers do more than enforce referential integrity.

  Because they are passive, constraints are limited to preventing updates within a narrow set of conditions. Triggers are more flexible. However, performance may suffer.

Index

I

K

M

N

O

P

R

S

T

U

W