Using Triggers

Version 5.x
December 2001
Part No. 000-7230A

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2001. All rights reserved.

US Government User Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Preface

Using Triggers is a complete guide to using triggers in the Informix implementation of Structured Query Language (SQL).

This user manual assumes that you have database management experience and are familiar with relational database concepts. It also assumes that you have knowledge of SQL and stored procedures. The Informix implementation of SQL is described in detail in a separate set of manuals called *The Informix Guide to SQL: Tutorial* and *The Informix Guide to SQL: Reference. The Informix Guide to SQL: Reference* also describes stored procedures and the Stored Procedure Language (SPL).

Using Triggers is both a user guide and a reference manual. The first chapter is an introduction to triggers. It tells you how to create and use triggers. The second chapter is a reference chapter. It describes both the syntax of the SQL statements for triggers and what happens when they execute.

Summary of Chapters

The *Using Triggers* user manual includes the following chapters:

- This Preface provides general information about the user manual and lists additional reference materials that help you understand concepts related to triggers.
- The Introduction describes how triggers fit into the Informix database server products, **INFORMIX-OnLine** and **INFORMIX-SE**. It explains how to use this user manual, describes other Informix product documentation, introduces the demonstration database from which the product examples are drawn, and describes the **Informix Messages and Corrections** product.
- Chapter 1, “An Introduction to Triggers,” provides an introduction to triggers. It explains how to create triggers and describes some of the uses for

triggers. It also illustrates and describes how to debug triggers by tracing them and how to generate error messages within triggers.

- Chapter 2, "A Reference for Triggers," describes the content of the system catalog tables for triggers and the syntax of the SQL statements that create and drop triggers. It also describes how triggers behave during execution and provides supporting examples.
- "Error Messages" contains a complete list of the error messages and corrective actions that are related to using triggers.
- The Index includes references throughout *Using Triggers.*

Related Reading

If you have prior experience with database management and are familiar with relational database concepts, but you would like more technical information, consider consulting the following texts by C. J. Date:

- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

This guide assumes you are familiar with your computer operating system. If you have limited UNIX system experience, you may want to look at your operating system manual or a good introductory text before starting to learn about triggers.

Some suggested texts about UNIX systems follow:

- *A Practical Guide to the UNIX System* by M. Sobell (Benjamin/Cummings Publishing, 1984)
- *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985)
- *UNIX for People* by Birns, Brown, and Muster (Prentice-Hall, 1985)

Table of Contents

Introduction
The Triggers Feature and Informix Products 3
Other Useful Documentation 4
How to Use This Manual 4
Typographical Conventions 5
Syntax Conventions 5
Example Code Conventions 9
Useful On-Line Files 10
ASCII and PostScript Error Message Files 10
Using the ASCII Error Message File 11
Using the PostScript Error Message Files 13
The Demonstration Database 13
Creating the Demonstration Database on INFORMIX-OnLine 14
Creating the Demonstration Database on INFORMIX-SE 15
Compliance with Industry Standards 16

Chapter 1 An Introduction to Triggers
Chapter Overview 1-3
An Overview of Triggers 1-3
Creating a Trigger 1-4
Components of the CREATE TRIGGER Statement 1-4
Creating a Trigger Using DB-Access 1-6
Creating a Trigger Using ESQL/C and ESQL/COBOL 1-7
Looking up a Trigger in the System Catalog 1-8
Using Triggers 1-10
Creating an Audit Trail 1-10
Implementing Business Rules 1-12

Deriving Additional Data 1-14
Enforcing Referential Integrity 1-16
Tracing Triggered Actions 1-17
Generating Error Messages 1-18

Chapter 2 **A Reference for Triggers**
Chapter Overview 2-3
System Catalog Tables for Triggers 2-3
SYSTRIGGERS 2-4
SYSTRIGBODY 2-5
SQL Statements for Triggers 2-6
CREATE TRIGGER 2-7
Purpose 2-7
Syntax 2-7
Usage 2-8
Trigger Name 2-11
UPDATE Clause 2-12
Action Clause 2-14
INSERT REFERENCING Clause 2-17
DELETE REFERENCING Clause 2-18
UPDATE REFERENCING Clause 2-19
Action Clause Subset 2-20
Triggered Action 2-21
Using Correlation Names in Triggered Actions 2-24
DROP TRIGGER 2-35
Purpose 2-35
Syntax 2-35
Usage 2-35
Triggers and Other SQL Statements 2-36

Appendix A **Notices**

Error Messages

Index

Introduction

The Triggers Feature and Informix Products 3

Other Useful Documentation 4

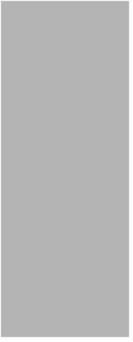
How to Use This Manual 4
 Typographical Conventions 5
 Syntax Conventions 5
 Example Code Conventions 9

Useful On-Line Files 10

ASCII and PostScript Error Message Files 10
 Using the ASCII Error Message File 11
  The **finderr** Script 11
  The **rofferr** Script 12
 Using the PostScript Error Message Files 13

The Demonstration Database 13
 Creating the Demonstration Database on INFORMIX-OnLine 14
 Creating the Demonstration Database on INFORMIX-SE 15

Compliance with Industry Standards 16



An SQL trigger is a mechanism that automatically sets off a specified set of SQL statements when a triggering event occurs on a table. It enables you to automate logically related changes to the database.

Triggers are stored in the database and, therefore, can reduce the amount of code that is required in programs that interact with the database. Two SQL statements, CREATE TRIGGER and DROP TRIGGER, allow you to create triggers on and drop triggers from tables, respectively. Two system catalog tables, **systriggers** and **systrigbody**, store information about the triggers in the database.

The Triggers Feature and Informix Products

Informix Software produces a variety of application development tools, CASE tools, database servers, and client/server products. Application development tools currently available include products like **INFORMIX-SQL**, **INFORMIX-4GL** and the **Interactive Debugger**, and the Informix embedded-language products, such as **INFORMIX-ESQL/C**.

Triggers are a feature of both the **INFORMIX-SE** and **INFORMIX-OnLine** database servers. You can use the **DB-Access** utility and the embedded-language products, **ESQL/C** and **ESQL/COBOL**, to create and use triggers. If you are running applications on a network, you will use an Informix client/server product such as **INFORMIX-NET** or **INFORMIX-STAR**. **INFORMIX-NET** is the communication facility for multiple **INFORMIX-SE** database servers. **INFORMIX-STAR** allows distributed database access to multiple **INFORMIX-OnLine** database servers.

Other Useful Documentation

You may want to refer to a number of related Informix product documents that complement *Using Triggers*.

- If you have never used Structured Query Language (SQL) or an Informix application development tool, read *The Informix Guide to SQL: Tutorial* to learn basic database design and implementation concepts.
- A companion volume to the Tutorial, *The Informix Guide to SQL: Reference,* provides full information on the structure and contents of the demonstration database that is provided with Informix products. It includes details of the Informix system catalog tables, describes Informix and common UNIX environment variables that should be set, and defines column data types supported by Informix products. Further, it provides a detailed description of all the SQL statements supported by Informix products. It also contains a glossary of useful terms.
- You, or whoever installs your Informix products, should refer to the *UNIX Products Installation Guide* for your particular release to ensure that your Informix product is properly set up before you begin to work with it.
- If you are using your database products across a network, you may also want to refer to the *INFORMIX-NET/INFORMIX-STAR Installation and Configuration Guide*.
- Depending on the database server you are using, you or your system administrator need either the *INFORMIX-OnLine Administrator's Guide* or the *INFORMIX-SE Administrator's Guide.*
- When errors occur, you can look them up, by number, and find their cause and solution in the*Informix Error Messages* manual. For error messages that are related to using triggers, see the "Error Messages" section of this manual. If you prefer, you can look up the error messages in the on-line message file described in the section "ASCII and PostScript Error Message Files" later in this Introduction.

How to Use This Manual

This section describes the typographical, syntax, and example code conventions used in *Using Triggers* and other Informix product documentation.

Typographical Conventions

The *Using Triggers* user manual uses a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout the manual:

italics	When new terms are introduced, they are printed in italics.
boldface	Database names, table names, column names, file names, utilities, and other similar terms are printed in boldface.
`computer`	Information that your Informix product displays and information that you enter are printed in a computer typeface.
KEYWORD	All keywords appear in uppercase letters.

Additionally, when you are instructed to “enter” or “execute” text, immediately press RETURN after the entry. When you are instructed to “type” the text, no RETURN is required.

Syntax Conventions

Syntax diagrams describe the format of SQL statements or commands, including alternative forms of a statement, required and optional parts of the statement, and so forth. Syntax diagrams have their own conventions, which are defined in detail and illustrated in this section. SQL statements are listed in their entirety in Chapter 7 of *The Informix Guide to SQL: Reference*, although some statements may appear in other manuals.

Each syntax diagram displays the sequences of required and optional elements that are valid in a statement. Briefly:

- All keywords are shown in uppercase letters for ease of identification, even though you need not enter them that way.
- Words for which you must supply values are in italics.

A diagram begins at the upper left with a keyword. It ends at the upper right with a vertical line. Between these points you can trace any path that does not stop or back up. Each path describes a valid form of the statement.

Along a path, you may encounter the following elements:

KEYWORD — You must spell a word in uppercase letters exactly as shown; however, you can use either uppercase or lowercase letters when you enter it.

(.,;+*-/) — Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.

" " — Double quotes are literal symbols that you must enter as shown. You can replace a pair of double quotes with a pair of single quotes, if you prefer. You cannot mix double and single quotes.

variable — A word in italics represents a value that you must supply. The nature of the value is explained immediately following the diagram unless the variable appears in a box. In that case, the page number of the detailed explanation follows the variable name.

[ADD Clause p. 7-14] — A reference in a box represents a subdiagram on the same page or another page. Imagine that the subdiagram is spliced into the main diagram at this point.

[Relational Operator *see* SQLR] — A reference to the SQLR represents an SQL statement or segment described in Chapter 7 of *The Informix Guide to SQL: Reference.* Imagine that the statement or segment is spliced into the main diagram at this point.

I4GL — A code in an icon is a signal warning you that this path is valid only for some products or under certain conditions. The codes indicate the products or conditions that support the path. The following codes are used:

- SE — This path is valid only for **INFORMIX-SE**.
- OL — This path is valid only for **INFORMIX-OnLine**.
- STAR — This path is valid only for **INFORMIX-STAR**.
- INET — This path is valid only for **INFORMIX-NET**.
- I4GL — This path is valid only for **INFORMIX-4GL**.
- ISQL — This path is valid only for **INFORMIX-SQL**.
- ESQL — This path is valid for SQL statements in all the following embedded language products: **INFORMIX-ESQL/C**, **INFORMIX-ESQL/COBOL**, and **INFORMIX-ESQL/FORTRAN**.
- E/C — This path is valid only for **INFORMIX-ESQL/C**.

E/CO This path is valid only for **INFORMIX-ESQL/COBOL**.

E/F This path is valid only for **INFORMIX-ESQL/ FORTRAN**.

DB This path is valid only for **DB-Access**.

SPL This path is valid only if you are using Informix Stored Procedure Language (SPL).

\+ This path is an Informix extension to ANSI standard SQL. If you initiate Informix extension checking and include this syntax branch, you receive a warning. If you set the DBANSIWARN environment variable, you receive the warnings at run time. To receive the warnings at compile time, compile with the **-ansi** flag.

— ALL — A shaded option is the default. Even if you do not explicitly type the option, it will be in effect unless you choose another option.

Syntax enclosed in a pair of arrows indicates that this is a subdiagram.

The vertical line is a terminator and indicates that the statement is complete.

IN NOT A branch below the main line indicates an optional path.

, variable A loop indicates a path that can be repeated.

1 column key A gate (1) in an option indicates that you can only use that option once, even though it is within a larger loop.

In Chapter 7 of *The Informix Guide to SQL: Reference*, icons that appear in the left margin indicate that the accompanying shaded text is valid only for some products or under certain conditions. In addition to the icons described in the preceding list, you may encounter the following icons in the left margin:

ANSI This icon indicates that the functionality described in the shaded text is valid only if your database is ANSI-compliant.

X/O This icon indicates that the functionality described in the shaded text conforms to X/Open standards for dynamic SQL. This functionality is available when you compile your embedded-language application with the **-xopen** flag.

Figure 1 shows the elements of a syntax diagram for the CREATE DATABASE statement.

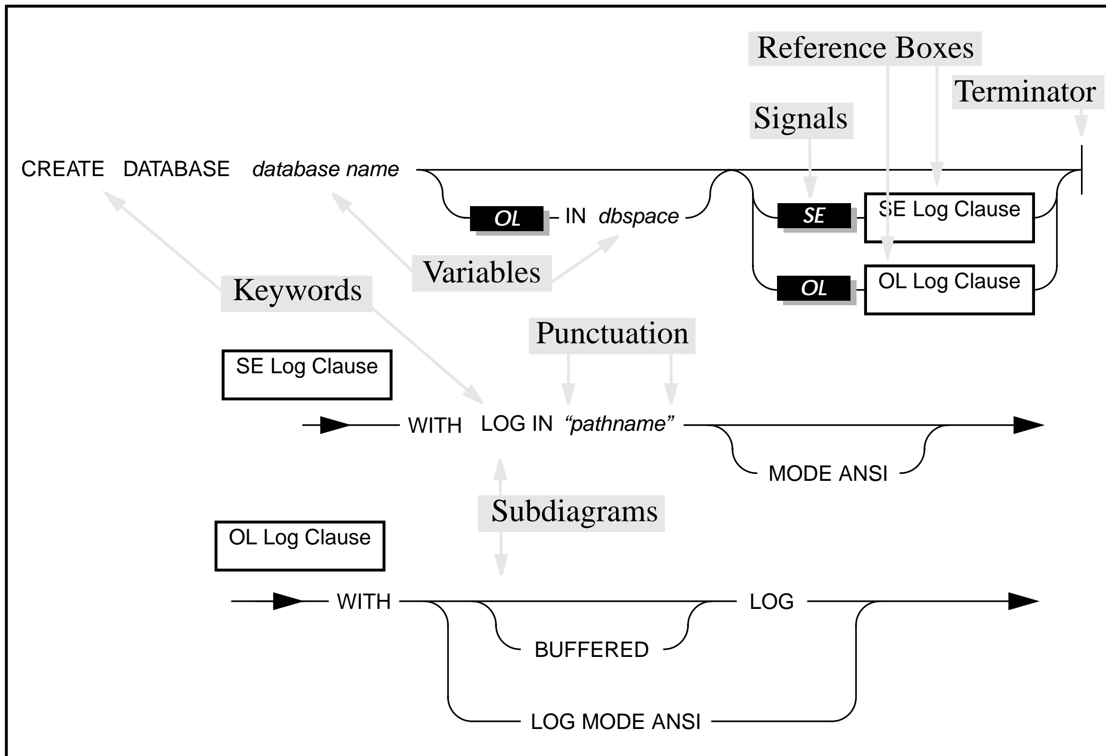


Figure 1 ***Elements of a syntax diagram***

To construct a statement using this diagram, start at the top left with the keywords CREATE DATABASE. Then follow the diagram to the right, proceeding through the options that you want. The diagram conveys the following information:

1. You must type the words CREATE DATABASE.
2. You must supply a *database name*.
3. You can stop, taking the direct route to the terminator, or you can take one or more of the optional paths.
4. If desired, you can designate a dbspace by typing the word IN and a dbspace name.

5. If desired, you can specify logging. Here, you are constrained by the database server with which you are working.
   - If you are using **INFORMIX-OnLine**, go to the subdiagram named *OL Log Clause*. Follow the subdiagram by typing the keyword WITH, then choosing and typing either LOG, BUFFERED LOG, or LOG MODE ANSI. Then, follow the arrow back to the main diagram.
   - If you are using **INFORMIX-SE**, go to the subdiagram named *SE Log Clause*. Follow the subdiagram by typing the keywords WITH LOG IN, typing a double quote, supplying a pathname, and closing the quotes. You can then choose the MODE ANSI option below the line or continue to follow the line across.
6. Once you are back at the main diagram, you come to the terminator. Your CREATE DATABASE statement is complete.

Example Code Conventions

Examples of SQL code appear throughout this user manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed, they are not delineated by semicolons. To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using **DB-Access**, you must delineate the statements with semicolons. If you are using an embedded language, you must use EXEC SQL and a semicolon (or other appropriate delimiters) at the start and end of each statement, respectively.

For example, you might see the following example code:

```
DATABASE stores5
.
.
.
DELETE FROM customer
    WHERE customer_num = 121
.
.
.
COMMIT WORK
CLOSE DATABASE
```

If you are using **DB-Access**, add semicolons at the end of each statement. If you are using **INFORMIX-ESQL/C**, add EXEC SQL or a dollar sign ($) at the beginning of each line and end each line with a semicolon. For detailed directions on using SQL statements for a specific application development tool, see the manual for your product.

Also note that ellipses in the example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

Useful On-Line Files

In addition to the Informix set of manuals, the following on-line files, located in the **$INFORMIXDIR/release** directory, may supplement the information in *Using Triggers*:

Documentation Notes	describe feature and performance topics not covered in the user manual or which have been modified since publication. The file containing the Documentation Notes for this feature is called **TRIGGERDOC_5.0**.
Release Notes	describe feature differences from earlier versions of Informix products and how these differences may affect current products. The file containing the Release Notes for this feature is called **ENGREL_5.0**.
Machine Notes	describe any special actions required to configure and use Informix products on your machine. The files containing the Machine Notes for this feature are called **SE_5.0** for **INFORMIX-SE** and **ONLINE_5.0** for **INFORMIX-OnLine**.

Please examine these files because they contain vital information about application and performance issues.

A number of Informix products also provide on-line Help files that walk you through each menu option. To invoke the Help feature in **DB-Access**, for example, simply press CTRL-W wherever you are in **DB-Access**.

ASCII and PostScript Error Message Files

Informix software products provide ASCII files that contain all the Informix error messages and their corrective actions. To access the error messages in the ASCII file, Informix provides scripts that let you display error messages on the terminal or print formatted error messages.

The optional **Informix Messages and Corrections** product provides PostScript files that contain the error messages and their corrective actions. If you install this product, you can print the PostScript files on a PostScript printer.

Using the ASCII Error Message File

You can use the file that contains the ASCII text version of the error messages and their corrective actions in two ways:

- Use the **finderr** script to display one or more error messages on the terminal screen.
- Use the **rofferr** script to print one error message or a range of error messages.

The scripts are in the **$INFORMIXDIR/bin** directory. The ASCII file has the following path:

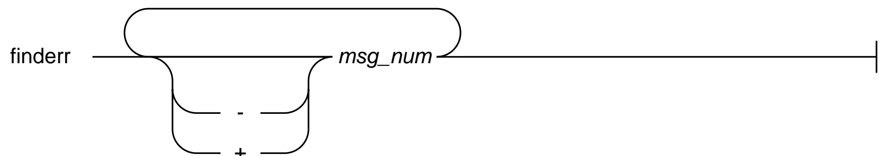
$INFORMIXDIR/msg/errmsg.txt

The error message numbers range from -1 to -33000. When you specify these numbers for the **finderr** or **rofferr** scripts, you can omit the minus sign. A few messages have positive numbers. In the event that you want to display them, you must precede the message number with a + sign.

The messages numbered -1 to -100 can be platform dependent. If the message text for a message in this range does not apply to your platform, check the operating system documentation for the precise meaning of the message number.

The *finderr* Script

Use the **finderr** script to display one or more error messages, and their corrective actions, on the terminal screen. The **finderr** script has the following syntax:



msg_num is the number of the error message to display.

You can specify any number of error messages per **finderr** command. The **finderr** command copies all the specified messages, and their corrective actions, to standard output.

For example, to display the -359 error message, you can enter the following command:

```
finderr -359
```

The following example demonstrates how to specify a list of error messages. This example also pipes the output to the UNIX **more** command to control the display. You can also redirect the output to another file so that you can save or print the error messages:

```
finderr 233 107 113 134 143 144 154 | more
```

The *rofferr* Script

Use the **rofferr** script to format one error message, or a range of error messages, for printing. By default, **rofferr** displays output on the screen. You need to send the output to **nroff** to interpret the formatting commands and then to a printer or to a file where the **nroff** output is stored until you are ready to print. You can then print the file. For information on using **nroff** and on printing files, see your UNIX documentation.

The **rofferr** script has the following syntax:

start_msg is the number of the first error message to format. This error message number is required.

end_msg is the number of the last error message to format. This error message number is optional. If you omit *end_msg*, only *start_msg* is formatted.

The following example formats error message -359. It pipes the formatted error message into **nroff** and sends the output of **nroff** to the default printer:

```
rofferr 359 | nroff -man | lpr
```

The following example formats and then prints all the error messages between -1300 and -4999:

```
rofferr -1300 -4999 | nroff -man | lpr
```

Using the PostScript Error Message Files

Use the **Informix Messages and Corrections** product to print the error messages and their corrective actions on a PostScript printer. The PostScript error messages are distributed in a number of files of the format **errmsg1.ps**, **errmess2.ps**, and so on. These files are located in the **$INFORMIXDIR/msg** directory.

The Demonstration Database

The **DB-Access** utility, provided with both the **INFORMIX-SE** and **INFORMIX-OnLine** database servers, includes a demonstration database called **stores5** that contains information about a fictitious wholesale sporting-goods distributor. The sample command files that make up a demonstration application are included as well.

Most of the examples in this user manual are based on the **stores5** demonstration database. The **stores5** database is described in detail and its contents are listed in Chapter 1 of *The Informix Guide to SQL: Reference.*

The script you use to install the demonstration database is called **dbaccessdemo5** and is located in the **$INFORMIXDIR/bin** directory. The database name that you supply is the name given to the demonstration database. If you do not supply a database name, the name defaults to **stores5**. Follow these rules for naming your database:

- Names for databases can be up to 10 characters long.
- The first character of a name must be a letter.
- You can use letters, characters, and underscores (_) for the rest of the name.
- **DB-Access** makes no distinction between uppercase and lowercase letters.
- The database name should be unique.

When you run **dbaccessdemo5**, you are, as the creator of the database, the owner and Database Administrator (DBA) of that database.

If you install your Informix database server product according to the installation instructions, the files that make up the demonstration database are protected so that you cannot make any changes to the original database.

You can run the **dbaccessdemo5** script again whenever you want to work with a fresh demonstration database. The script prompts you when the creation of the database is complete, and asks if you would like to copy the sample command files to the current directory. Answer "N" to the prompt if you have made changes to the sample files and do not want them replaced with the original versions. Answer "Y" to the prompt if you want to copy over the sample command files.

Creating the Demonstration Database on INFORMIX-OnLine

Use the following steps to create and populate the demonstration database in the **INFORMIX-OnLine** environment:

1. Set the INFORMIXDIR environment variable so that it contains the name of the directory in which your Informix products are installed. Set SQLEXEC to **$INFORMIXDIR/lib/sqlturbo**. (For a full description of environment variables, see Chapter 4 of *The Informix Guide to SQL: Reference.*)
2. Create a new directory for the SQL command files. Create the directory by entering

   ```
   mkdir dirname
   ```

3. Make the new directory the current directory by entering

   ```
   cd dirname
   ```

4. Create the demonstration database and copy over the sample command files by entering

   ```
   dbaccessdemo5 dbname
   ```

   The data for the database is put into the root dbspace.

To give someone else the SQL privileges to access the data, use the GRANT and REVOKE statements. The GRANT and REVOKE statements are described in Chapter 7 of *The Informix Guide to SQL: Reference.*

To use the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **dbaccessdemo5** script. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command.

Creating the Demonstration Database on INFORMIX-SE

Use the following steps to create and populate the demonstration database in the **INFORMIX-SE** environment:

1. Set the INFORMIXDIR environment variable so that it contains the name of the directory in which your Informix products are installed. Set SQLEXEC to **$INFORMIXDIR/lib/sqlexec**. (For a full description of environment variables, see Chapter 4 of *The Informix Guide to SQL: Reference.*)
2. Create a new directory for the demonstration database. This directory will contain the example command files included with the demonstration database. Create the directory by entering

   ```
   mkdir dirname
   ```

3. Make the new directory the current directory by entering

   ```
   cd dirname
   ```

4. Create the demonstration database and copy over the sample command files by entering

   ```
   dbaccessdemo5 dbname
   ```

When you run the **dbaccessdemo5** script, it creates a subdirectory called *dbname*.**dbs** in your current directory and places the database files associated with **stores5** there. You will see both data and index files in the *dbname*.**dbs** directory.

To use the database and the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **dbaccessdemo5** script. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command. Check with your system administrator for more information about operating system file and directory permissions. UNIX permissions are discussed in the *INFORMIX-SE Administrator's Guide.*

To give someone else access to the database that you have created, grant them the appropriate privileges using the GRANT statement in **DB-Access**. To remove privileges, use the REVOKE statement. The GRANT and REVOKE statements are described in Chapter 7 of *The Informix Guide to SQL: Reference.*

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are compliant with ANSI Level 2 (published as ANSI X3.135-1989) on the **INFORMIX-OnLine** database server. They are compliant with ANSI Level 2 on the **INFORMIX-SE** database server with the following exceptions:

- Effective checking of constraints
- Serializable transactions

INFORMIX-TP/XA conforms to the *X/Open Preliminary Specification (April 1990), Distributed Transaction Processing: The XA Interface.*

Chapter

1

An Introduction to Triggers

Chapter Overview 3

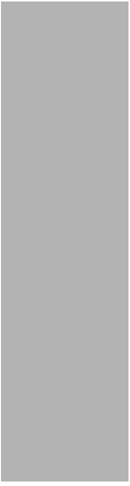
An Overview of Triggers 3

Creating a Trigger 4
- Components of the CREATE TRIGGER Statement 4
  - Trigger Name 4
  - Trigger Event 4
  - REFERENCING Clause 5
  - Action Clause 5
- Creating a Trigger Using DB-Access 6
  - Adding Comments in DB-Access 6
- Creating a Trigger Using ESQL/C and ESQL/COBOL 7
- Looking up a Trigger in the System Catalog 8

Using Triggers 10
- Creating an Audit Trail 10
- Implementing Business Rules 12
- Deriving Additional Data 14
- Enforcing Referential Integrity 16

Tracing Triggered Actions 17

Generating Error Messages 18



Chapter Overview

This chapter is an introduction to creating and using SQL triggers in Informix databases. It tells you what a trigger is and how to create one using either the **DB-Access** utility or one of the Informix embedded-language products. The chapter provides several examples that illustrate some of the uses for triggers, the use of stored procedures as triggered actions, and the ability to cascade triggers. This chapter also shows you how to debug triggers and how to generate error messages inside triggered actions.

An Overview of Triggers

An SQL *trigger* is a mechanism that automatically sets off a specified set of SQL statements when a triggering event occurs on a table. For example, when you insert a row into a table of order items, you might want to calculate the total price of all items that the customer has ordered to see if the total price exceeds the customer's credit limit. Or when you delete an item from the table of items, you might want to delete the corresponding order from the table of orders if all the items for it have been deleted. Triggers enable you to automate these types of procedures.

Basically, a trigger consists of a *trigger event* and a resulting *triggered action.* The trigger event can be an INSERT or DELETE statement, or it can be an UPDATE statement that updates one or more columns that you specify as triggering columns in a table. The triggered action is the set of SQL statements that are executed when the trigger event occurs. The triggered action can consist of INSERT, DELETE, UPDATE, and EXECUTE PROCEDURE statements. See *The Informix Guide to SQL: Reference* for more information about these SQL statements.

A trigger is stored as an object in the database. Any user who has the required privilege can use it. As a result, triggers can reduce the amount of code that is required in applications that access the database. Triggers can also prevent redundant code that would otherwise be required when multiple programs perform the same operations on the database.

Among other uses, you can use triggers to create an audit trail, implement business rules, derive additional data, and enforce referential integrity. See “Using Triggers” on page 1-10 for examples of these uses.

Creating a Trigger

You use the CREATE TRIGGER statement to create a trigger on a table. You can create a trigger using either **DB-Access**, **INFORMIX-ESQL/C**, or **INFORMIX-ESQL/COBOL**.

Components of the CREATE TRIGGER Statement

In the CREATE TRIGGER statement, you define the following elements:

- The trigger name
- The trigger event
- The optional REFERENCING clause
- The action clause

Figure 1-1 illustrates the CREATE TRIGGER statement, showing each of these elements on a separate line.

```
CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE items_pct (post_ins.manu_code))
```

Figure 1-1 Components of the CREATE TRIGGER statement

Trigger Name

The first part of the CREATE TRIGGER statement assigns a name to the trigger. The name of the trigger follows the keywords CREATE TRIGGER. In Figure 1-1, the name of the trigger is **items_ins**.

Trigger Event

The second part of the CREATE TRIGGER statement specifies the trigger event. In Figure 1-1, the trigger event is an INSERT on the **items** table. See *The Informix Guide to SQL: Reference* for a complete description of the tables in the **stores5** demonstration database.

REFERENCING Clause

The third part of the CREATE TRIGGER statement is the optional REFERENCING clause. You can only use the REFERENCING clause with an action clause that begins with the keywords FOR EACH ROW, as in Figure 1-1. (See the next section "Action Clause" for the meaning of the keywords FOR EACH ROW.)

The REFERENCING clause lets you define two prefixes that you can use in the action clause with a column name from the triggering table. One prefix refers to the value of a column before the triggering statement takes effect; the other prefix refers to the value of a column after the triggering statement takes effect. If the triggering statement is an UPDATE, for example, you might want to refer in the action clause to either the old or new value of the column in the current row. You can refer to either value by defining a prefix for it.

In Figure 1-1, the REFERENCING clause uses the keyword NEW to define a new prefix name called **post_ins**. A new prefix name refers to column values after the triggering statement takes effect. In the action clause, **post_ins** precedes the column name **manu_code** to refer to the value of that column after the triggering insert is complete. To define a prefix name that refers to column values before the triggering statement takes effect, precede the prefix name in the REFERENCING clause with the keyword OLD.

In the remainder of this user manual, the old and new prefix names are called *correlation names*. See "Using Correlation Names in Triggered Actions" on page 2-24 for more information on the proper use of these names.

Action Clause

The fourth, and last, part of the CREATE TRIGGER statement is the action clause. The action clause consists of the following two parts:

- Keywords that specify when the action occurs, relative to the triggering statement
- Triggered action lists that contain the SQL statements to be executed.

In Figure 1-1, the keywords FOR EACH ROW specify that the triggered action that follows occurs once for each row that the triggering statement inserts. In this case, the triggered action list consists of a single EXECUTE PROCEDURE statement that calls the stored procedure **items_pct**.

You can also define triggered action lists that are preceded by the keywords BEFORE and AFTER. The keyword BEFORE tells the database server to execute the statements in the triggered action list before it executes the triggering statement. The keyword AFTER tells the database server to execute the statements in the triggered action list after the triggering statement is complete.

BEFORE and AFTER triggered actions execute only once, whereas FOR EACH ROW triggered actions execute for each row that the triggering statement inserts, deletes, or updates.

In Figure 1-1, for each row that the database server inserts into the **items** table, it subsequently executes the **items_pct** stored procedure.

Creating a Trigger Using DB-Access

You create a trigger using **DB-Access** by selecting New on the QUERY-LANGUAGE Menu and typing the CREATE TRIGGER statement under the SQL editor, as shown in Figure 1-2.

```
NEW:    ESC    = Done editing      CTRL-A = Typeover/Insert     CTRL-R = Redraw
        CTRL-X = Delete character  CTRL-D = Delete rest of line

----------------------- stores5 ----------------- Press CTRL-W for Help ---------

CREATE TRIGGER items_ins INSERT ON items REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE items_pct (post_ins.manu_code;))
```

Figure 1-2 ***Entering a CREATE TRIGGER statement using*** **DB-Access**

Adding Comments in DB-Access

When you use **DB-Access** to create a trigger, you can include comments with the CREATE TRIGGER statement. To add a comment, either enclose it between braces ({}) or precede it with two dashes (--). The use of two dashes is the ANSI-compliant method of introducing a comment. Figure 1-3 illustrates both types of comments.

```
SQL:   New [Run] Modify  Use-editor  Output  Choose  Save  Info  Drop  Exit
Run the current SQL statements.

----------------------- stores5 ---------------- Press CTRL-W for Help --------

-- This ANSI-compliant comment is not preserved in systrigbody
CREATE TRIGGER del_stock
DELETE ON stock
{ for each delete, delstock_p1() counts it }
FOR EACH ROW(EXECUTE PROCEDURE delstock_p1())
-- if any deletes, delstock_p2 makes an entry in stock_log
AFTER (EXECUTE PROCEDURE delstock_p2());
```

Figure 1-3 ***ANSI-compliant and non-ANSI-compliant comments in a CREATE TRIGGER statement***

When the database server processes a CREATE TRIGGER statement, it stores the definition of the trigger in the **systrigbody** system catalog table. Any comments that you place before or after the CREATE TRIGGER statement are not preserved when the database server stores the trigger definition. The database server only preserves comments that appear within the CREATE TRIGGER statement. See "System Catalog Tables for Triggers" on page 2-3 for a description of the **systrigbody** system catalog table.

Creating a Trigger Using ESQL/C and ESQL/COBOL

You can embed the CREATE TRIGGER statement in an **INFORMIX-ESQL/C** or **INFORMIX-ESQL/COBOL** program by following the product conventions for embedding SQL statements. Following the ANSI standard, you embed an SQL statement in an **ESQL/C** source program by preceding it with the keywords EXEC SQL, as shown in Figure 1-4.

```
#include <stdio.h>

main()
{
.
.
.
    EXEC SQL DATABASE stores5;
    EXEC SQL CREATE TRIGGER items_ins INSERT ON items
        REFERENCING NEW AS post_ins
        FOR EACH ROW(EXECUTE items_pct (post_ins.manu_code));
.
.
.
```

Figure 1-4 ***A CREATE TRIGGER statement in an*** INFORMIX-ESQL/C ***program***

To embed a CREATE TRIGGER statement in an **ESQL/COBOL** program, place the statement in the procedure division and enclose it between the phrases EXEC SQL and END-EXEC. Figure 1-5 shows you how to embed the CREATE TRIGGER statement in an **ESQL/COBOL** program.

```
PROCEDURE DIVISION.
MAIN.
.
.
.
EXEC SQL DATABASE STORES5 END-EXEC. --open stores5 database
EXEC SQL CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE items_pct (post_ins.manu_code)) END-EXEC.
.
.
.
```

Figure 1-5 ***A CREATE TRIGGER statement in an*** **INFORMIX-ESQL/COBOL** ***program***

Note: *When you use an embedded-language product to create a trigger within a program, the comments within the CREATE TRIGGER statement are not preserved. The embedded-language preprocessor strips the comments from the program before the database server stores the definition in the* ***systrigbody*** *table.*

Looking up a Trigger in the System Catalog

The system catalog tables **systriggers** and **systrigbody** support triggers. See "System Catalog Tables for Triggers" on page 2-3 for a description of the content of these tables. See Chapter 2 of *The Informix Guide to SQL: Reference* for a description of the purpose and content of the system catalog. You can query the **systriggers** and **systrigbody** tables, just as you would any other table, to obtain information about triggers in the database.

The following SELECT statement queries the **systrigbody** table for the sequence number (**seqno**) and header information (**datakey** = 'D') on a trigger named **upqty_i.** The query selects **seqno** and orders the result on this column in case the header consists of more than one row. The query uses a

subquery on **systriggers** to obtain the value of **trigid** for this trigger. For more information on the **systrigbody** and **systriggers** system catalog tables, see “System Catalog Tables for Triggers” on page 2-3.

```
SELECT seqno, data FROM systrigbody WHERE datakey = 'D'
    AND trigid = (SELECT trigid FROM systriggers
        WHERE trigname = "upqty_i")
    ORDER BY seqno
```

The **DB-Access** screen shown in Figure 1-6 displays the result of this query. The header information for a trigger consists of the trigger name, the triggering statement, and the REFERENCING clause.

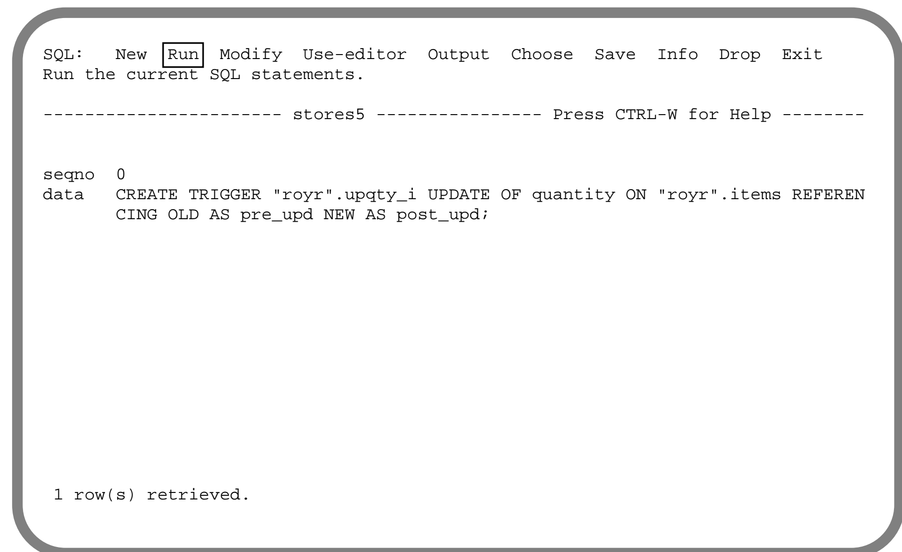


Figure 1-6 ***Result of a query on the* systrigbody *table for a trigger header***

Using Triggers

Triggers let you automate changes to the database. With triggers, an insert, delete, or update on a table can set off additional changes to the database to enforce various rules or practices. Triggers can enforce accounting rules, automate departmental practices, perform calculations, maintain the integrity of the database, and so on. To illustrate some uses for triggers, this section provides examples of the following applications:

- Creating an audit trail
- Implementing business rules
- Deriving additional data
- Enforcing referential integrity

Creating an Audit Trail

Perhaps the simplest application for a trigger is to create an audit trail of activity that occurs in the database. The statements in Figure 1-7 create a table called **log_record** and a trigger called **upqty_i**.

```
CREATE TABLE log_record
    (item_num    SMALLINT,
    ord_num      INTEGER,
    username     CHARACTER(8),
    update_time  DATETIME YEAR TO MINUTE,
    old_qty      SMALLINT,
    new_qty      SMALLINT);

CREATE TRIGGER upqty_i
UPDATE OF quantity ON items
REFERENCING OLD AS pre_upd
        NEW AS post_upd
FOR EACH ROW(INSERT INTO log_record
    VALUES (pre_upd.item_num, pre_upd.order_num, USER, CURRENT,
            pre_upd.quantity, post_upd.quantity));
```

Figure 1-7 A trigger that audits changes to a table

An update of the **quantity** column in the **items** table activates the **upqty_i** trigger. The action clause specifies FOR EACH ROW, which means that the triggered action occurs once for each row that the triggering statement updates in the **items** table. After each row is updated, the triggered action inserts a row into the **log_record** table, which stores the values for the **item_num** and **order_num** columns from the updated row. It also stores the

user's name, the time of the update, and the old and new values for **quantity**. An UPDATE statement like the one in the following example activates **upqty_i**.

```
UPDATE items SET quantity = quantity * 2 WHERE order_num = 1007
    AND manu_code = 'HRO'
```

When this UPDATE statement triggers **upqty_i** , the triggered INSERT statement inserts four rows in the **log_record** table, as shown in Figure 1-8.

item_num	ord_num	username	update_time	old_qty	new_qty
1	1007	dale	1992-07-30 12:51	1	2
2	1007	dale	1992-07-30 12:51	1	2
4	1007	dale	1992-07-30 12:51	1	2
5	1007	dale	1992-07-30 12:51	1	2

Figure 1-8 ***Rows inserted to the*** **log_record** ***table by the*** **upqty_i** ***trigger***

Implementing Business Rules

You can also use triggers to implement business rules. A business rule is a practice that an organization enforces as a matter of doing business. A merchant, for example, might reject any order that exceeds a customer's credit limit. Or a corporation might match a portion of an employee's contributions to the retirement fund after the employee completes one year of service.

In Figure 1-9, the trigger **up_items** enforces a rule that says no single update to the **items** table shall increase the total quantity on order for all items by more than 50 percent.

```
CREATE PROCEDURE upd_items_p1()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    LET old_qty = (SELECT SUM(quantity) FROM items);
END PROCEDURE;

CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -746, 0, "Not allowed - rule violation;"
    END IF
END PROCEDURE;

CREATE TRIGGER up_items
UPDATE OF quantity ON items
BEFORE(EXECUTE PROCEDURE upd_items_p1())
AFTER(EXECUTE PROCEDURE upd_items_p2());
```

Figure 1-9 ***A trigger that implements a business rule***

To determine the effect of the triggering UPDATE statement, the **up_items** trigger uses BEFORE and AFTER action clauses to construct before and after images of the **items** table. The BEFORE action clause executes the stored procedure **upd_items_p1**, which calculates the total quantity on order for all items before the triggering statement executes. After the triggering UPDATE statement completes, the AFTER action clause executes the stored procedure **upd_items_p2**, which performs the same calculation again. This time, however, the result includes the quantities that have just been updated. If the total quantity for all items after the update is more than 50 percent greater than the total quantity before the update, **upd_items_p2** uses the RAISE EXCEPTION statement to generate error -746 and display the following message:

```
Not allowed - rule violation.
```

When a trigger fails in **INFORMIX-OnLine**, if the database has logging, the database server rolls back the changes made by both the triggering statement and the triggered actions. See “Logging and Recovery” on page 2-33 for more information on the interaction of triggers and logging.

Figure 1-10 illustrates the outcome when an undesirable UPDATE statement triggers **up_items.**

```
SQL:   New  Run  Modify  Use-editor  Output  Choose  Save  Info  Drop  Exit
Modify the current SQL statements using the SQL editor.

----------------------- stores5 --------------- Press CTRL-W for Help --------

UPDATE items SET quantity = quantity * 3 WHERE manu_code = "ANZ";


   746: Not allowed - rule violation
```

Figure 1-10 ***Outcome of the*** **up_items** ***trigger on an undesirable update***

Deriving Additional Data

You can use a trigger to derive data that is not directly available from the triggering table. In Figure 1-11, when an update occurs on the column **quantity** in the **items** table, the trigger **uptot_pr** executes the stored procedure **calc_totpr** to calculate the corresponding adjustment to the **total_price** column. The **calc_totpr** procedure performs two calculations to derive the amount of the adjustment to **total_price**. Using local procedure variables and the LET statement from the Stored Procedure Language (SPL), the procedure divides the old total price for the item by the old value of **quantity** to obtain the unit price. It then multiplies the unit price by the new value of **quantity** to obtain the new total price. See Chapter 8 of *The Informix Guide to SQL: Reference* for a description of SPL statements.

```
CREATE PROCEDURE calc_totpr(old_qty SMALLINT, new_qty SMALLINT,
    total MONEY(8)) RETURNING MONEY(8);
    DEFINE u_price LIKE items.total_price;
    DEFINE n_total LIKE items.total_price;
    LET u_price = total / old_qty;
    LET n_total = new_qty * u_price;
    RETURN n_total;
END PROCEDURE;

CREATE TRIGGER upd_totpr
UPDATE OF quantity ON items
REFERENCING OLD AS pre_upd
    NEW AS post_upd
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
    post_upd.quantity, pre_upd.total_price) INTO total_price);
```

Figure 1-11 A trigger that derives additional data

When you use a stored procedure as a triggered action in a FOR EACH ROW action clause, you can use the INTO clause of the EXECUTE PROCEDURE statement to update nontriggering columns in the current row of the triggering table with values returned by the procedure. Outside of a triggered action, the syntax of the EXECUTE PROCEDURE statement restricts you to using only embedded language host variables in the INTO clause. Inside a triggered action, however, you can name columns from the triggering table in the INTO clause. In a triggered action, use of the INTO clause in the EXECUTE PROCEDURE statement implies that the columns named are updated with values that the procedure returns. In the example, the stored procedure **calc_totpr** returns **n_total**, which is updated into the **total_price** column of the current row in the **items** table.

Using the EXECUTE PROCEDURE statement as a triggered action also enables you to pass data from the triggering table to the triggered stored procedure. In the example, the EXECUTE PROCEDURE statement passes the old and new values for **quantity** and the old value for **total_price** to the stored procedure **calc_totpr**. See Chapter 7 of *The Informix Guide to SQL: Reference* for a description of the EXECUTE PROCEDURE statement.

The following query displays four rows from the **items** table:

```
SELECT * FROM items WHERE order_num = 1007 AND manu_code = "HRO"
```

Prior to an update that triggers **upd_totpr**, these four rows appear as shown in Figure 1-12:

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1007	1	HRO	1	$250.00
2	1007	2	HRO	1	$126.00
4	1007	4	HRO	1	$480.00
5	1007	7	HRO	1	$600.00

Figure 1-12 Result of query before update

An UPDATE statement that doubles the value of **quantity** for these four rows also triggers **upd_totpr**, which produces a corresponding increase in the **total_price** column. Figure 1-13 shows the result of both the update and the trigger for these same four rows.

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1007	1	HRO	2	$500.00
2	1007	2	HRO	2	$252.00
4	1007	4	HRO	2	$960.00
5	1007	7	HRO	2	$1200.00

***Figure 1-13 Result of query after an update on quantity has triggered* upd_totpr**

Enforcing Referential Integrity

You can use triggers to enforce referential integrity in the database. Referential integrity is the dependency of data in one table on data in another table. For example, in the **stores5** database the column **customer_num** is the primary key in the **customer** table, but it also appears as a foreign key in both the **orders** and **cust_calls** tables. This relationship means that the data in these latter tables is associated with particular customers in the **customer** table. So if you delete a customer from the **customer** table, to maintain the integrity of your data, you must also delete the data for this customer from the **orders** and **cust_calls** tables. Otherwise one of the values in the **customer_num** column of these tables is no longer associated with a customer. A similar relationship exists in the **stores5** database between the **orders** table and the **items** table. For a complete description of referential integrity in the **stores5** database, see Chapter 1 of *The Informix Guide to SQL: Reference.*

For example, assume that a customer goes out of business and you delete that customer from the **customer** table. In Figure 1-14, the triggers **del_cust** and **del_items** perform the deletes that are required to maintain data integrity in the **orders**, **cust_calls**, and **items** tables when you delete a customer.

```
CREATE TRIGGER del_cust
DELETE ON customer
REFERENCING OLD AS pre_del
FOR EACH ROW(DELETE FROM orders WHERE customer_num =
        pre_del.customer_num,
    DELETE FROM cust_calls WHERE customer_num =
        pre_del.customer_num);

CREATE TRIGGER del_items
DELETE ON orders
REFERENCING OLD AS pre_del
FOR EACH ROW(DELETE FROM items WHERE order_num =
        pre_del.order_num);
```

Figure 1-14 Triggers to enforce referential integrity when a customer is deleted

In an **INFORMIX-OnLine** database with logging, when a trigger executes, the database server does not enforce referential constraints until after the triggered action is complete. This allows the triggered action to rectify any constraint violations created by the triggering statement. In an **INFORMIX-SE** database, however, when the triggering statement violates a referential con-

straint, the database server returns the error before the triggered action executes. See "Constraint Checking" on page 2-30 for more information on how the database server checks constraints when it executes a trigger.

Notice in Figure 1-14 that the two triggers cascade, meaning that the action of one trigger sets off another one. **INFORMIX-OnLine** and **INFORMIX-SE** both allow triggers to cascade up to a maximum of 61 triggers in a series, including the initial trigger. In Figure 1-14, the delete on the **orders** table in the first trigger, **del_cust**, sets off the second trigger, **del_items**. See "Cascading Triggers" on page 2-29 for more information on cascading triggers.

Tracing Triggered Actions

If you find that a triggered action is not behaving as you expect, you can monitor its execution by placing it inside a stored procedure and using the TRACE statement in SPL. In Figure 1-15, TRACE statements have been added to the stored procedure **items_pct.** Before starting the trace, you must direct the output to a file with the SET DEBUG FILE TO statement. In Figure 1-15, the SET DEBUG FILE TO statement directs the trace output to the file **/usr/mydir/trig.trace**. The TRACE ON statement begins tracing the procedure

statements and variables. See Chapter 8, "Stored Procedures and SPL," in *The Informix Guide to SQL: Reference* for more information on tracing stored procedures and using the TRACE statement.

```
CREATE PROCEDURE items_pct(mac CHAR(3))
DEFINE tp MONEY;
DEFINE mc_tot MONEY;
DEFINE pct DECIMAL;
SET DEBUG FILE TO "/usr/mydir/trig.trace";
TRACE "begin trace";
TRACE ON
LET tp = (SELECT SUM(total_price) FROM items);
LET mc_tot = (SELECT SUM(total_price) FROM items
    WHERE manu_code = mac);
LET pct = mc_tot / tp;
IF pct > .10 THEN
    RAISE EXCEPTION -745;
END IF
TRACE OFF;
END PROCEDURE;

CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE items_pct (post_ins.manu_code));
```

Figure 1-15 Tracing the triggered action in a stored procedure

Figure 1-16 shows the trace output from the **items_pct** procedure to the file **/usr/mydir/trig.trace**. The trace output reveals the values of procedure variables, procedure arguments, return values, and error codes.

```
trace expression :begin trace
trace on
expression:
  (select (sum total_price)
    from items)
evaluates to $18280.77 ;
let  tp = $18280.77
expression:
  (select (sum total_price)
    from items
    where (= manu_code, mac))
evaluates to $3008.00 ;
let  mc_tot = $3008.00
expression:(/ mc_tot, tp)
evaluates to 0.16
let  pct = 0.16
expression:(> pct, 0.1)
evaluates to 1
expression:(- 745)
evaluates to -745
raise exception :-745, 0, ""
exception : looking for handler
SQL error = -745 ISAM error = 0  error string =  = ""
exception : no appropriate handler
```

Figure 1-16 ***Trace output from the*** **items.pct** ***stored procedure***

Generating Error Messages

When a trigger fails because of an SQL statement, the database server returns the applicable SQL error. New SQL error messages that are specifically related to triggers, and error messages that have been changed to pertain to triggers, are provided in the “Error Messages” section of this manual, along with their corrective actions.

In a triggered action that is a stored procedure, you can originate an error message to apply to a condition that you specify. Two error numbers are reserved for use with triggers. The first one is error number -745, which has a generalized and fixed error message. The second one is error number -746, for which you must supply message text, up to a maximum of 71 characters.

You can apply error number -745 to any trigger failure that is not an SQL error. The fixed message for this error is as follows:

```
-745 Trigger execution has failed.
```

You can generate this message by using the RAISE EXCEPTION statement in SPL. The following example generates error -745 if **new_qty** is greater than **old_qty** multiplied by 1.50.

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -745;
    END IF
END PROCEDURE
```

If you are using the **DB-Access** utility, the text of the message for error -745 displays on the bottom of the screen, as seen in Figure 1-17.

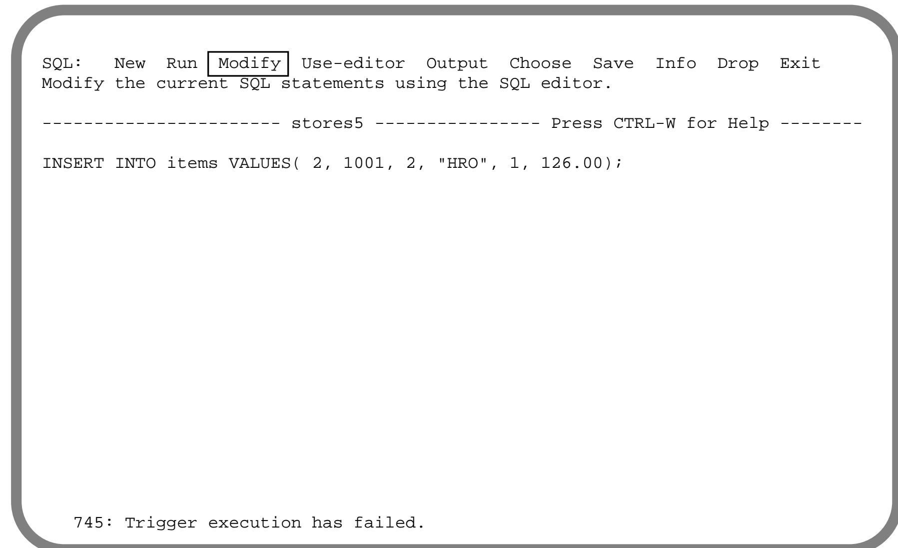


Figure 1-17 Error message -745 with fixed message text

If you trigger the erring procedure through an SQL statement in your embedded-language program, the database server sets the SQL error status variable to -745 and returns it to your program. To display the text of the message, follow the procedure that your Informix application development tool provides for retrieving the text of any SQL error message.

Error number -746 requires you to provide the text of the message. The following example again generates an error if **new_qty** is greater than **old_qty** multiplied by 1.50. However, this time the error number is -746 and the message text, `Too many items for Mfr.`, is supplied as the third argument in the RAISE EXCEPTION statement. See the RAISE EXCEPTION statement in Chapter 8 of the *The Informix Guide to SQL: Reference*.

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -746, 0, "Too many items for Mfr.";
    END IF
END PROCEDURE
```

If you use **DB-Access** to submit the triggering statement, and if **new_qty** is greater than **old_qty**, the result of this stored procedure is shown in Figure 1-18.

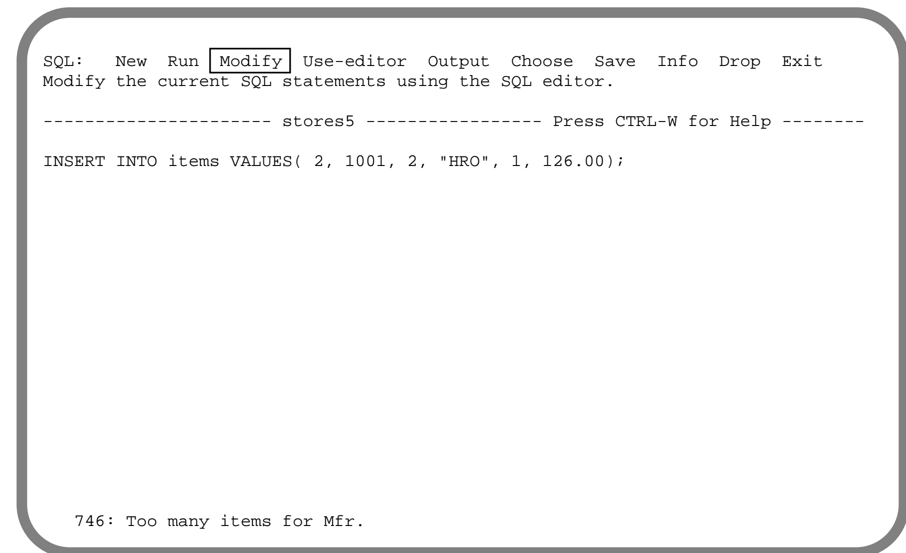


Figure 1-18 Error number -746 with user-specified message text

If you trigger the stored procedure through an SQL statement in your embedded-language program, the database server sets the SQL error status variable to -746 and returns the message text in the **sqlerrm** field of the SQL Communications Area (SQLCA). See Chapter 6 of *The Informix Guide to SQL: Tutorial* or the manual for your embedded-language product for a description of the SQLCA.

Chapter

2

A Reference for Triggers

Chapter Overview 3

System Catalog Tables for Triggers 3
- SYSTRIGGERS 4
- SYSTRIGBODY 5

SQL Statements for Triggers 6

CREATE TRIGGER 7
- Purpose 7
- Syntax 7
- Usage 8
  - The Trigger Event 8
  - Impact of Triggers 10
- Trigger Name 11
- UPDATE Clause 12
  - Defining Multiple Update Triggers 12
  - When an UPDATE Statement Activates Multiple Triggers 13
- Action Clause 14
  - BEFORE Actions 14
  - FOR EACH ROW Actions 14
  - AFTER Actions 15
  - Actions of Multiple Triggers 15
  - Guaranteeing Row-Order Independence 15
- INSERT REFERENCING Clause 17
- DELETE REFERENCING Clause 18
- UPDATE REFERENCING Clause 19
- Action Clause Subset 20

Triggered Action 21
The WHEN Condition 21
The Action Statements 22
Using Correlation Names in Triggered Actions 24
When to Use Correlation Names 24
Qualified Versus Unqualified Value 25
Rules for Stored Procedures 27
Privileges to Execute Triggered Actions 28
Cascading Triggers 29
Constraint Checking 30
Preventing Triggers from Overriding Each Other 31
The Client/Server Environment 32
Logging and Recovery 33
DROP TRIGGER 35
Purpose 35
Syntax 35
Usage 35
Triggers and Other SQL Statements 36

Chapter Overview

This chapter describes the **systriggers** and **systrigbody** system catalog tables and the two SQL statements CREATE TRIGGER and DROP TRIGGER. It provides both syntax and usage rules for these statements.

The chapter also includes information on the following topics related to triggers:

- Trigger events with cursors
- Privileges required to create and execute triggers
- Multiple update trigggers
- Rules for stored procedures
- Cascading triggers
- Constraint checking
- Client/server environment
- Logging and recovery
- Triggers and other SQL statements

System Catalog Tables for Triggers

Two system catalog tables, **systriggers** and **systrigbody**, support triggers. The system catalog consists of tables that describe the structure of the database. Each table contains specific information about an element in the database. For example, the system catalog tracks the views, authorized users, and privileges associated with every table you create.

The system catalog tables are generated automatically when you create a database, and you can query them as you would query any other table in the database. For more information about the system catalog tables, see Chapter 2 of *The Informix Guide to SQL: Reference*.

SYSTRIGGERS

The **systriggers** system catalog table contains miscellaneous information about the trigger, including the trigger event and the correlated reference specification. The **systriggers** system catalog table has the following columns:

Column Name	Type	Explanation
trigid	SERIAL	trigger ID
trigname	CHAR(18)	trigger name
owner	CHAR(8)	owner of trigger
tabid	INT	ID of triggering table
event	CHAR	triggering event: I insert trigger U update trigger D delete trigger
old	CHAR(18)	name of value before update
new	CHAR(18)	name of value after update
mode	CHAR	(reserved for future use)

A composite index for the **trigname** and **owner** columns allows only unique values. The **trigid** column is indexed and must contain unique values. An index for the **tabid** column allows duplicate values.

If REFERENCING is specified in the trigger, the old correlation name is stored in the field **old** and the new correlation name is stored in the field **new**. See "INSERT REFERENCING Clause" on page 2-17, "DELETE REFERENCING Clause" on page 2-18, and "UPDATE REFERENCING Clause" on page 2-19 for more information on the correlated-reference specification.

SYSTRIGBODY

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

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

The **systrigbody** system catalog table has the following columns:

Column Name	Type	Explanation
trigid	INT	trigger ID
datakey	CHAR	type of data: D English text for the header, trigger definition A English text for the body, triggered actions H linearized code for the header S linearized code for the symbol table B linearized code for the body
seqno	INT	sequence number
data	CHAR(256)	English text or linearized code

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

SQL Statements for Triggers

Use the following SQL data definition statements to create and drop triggers:

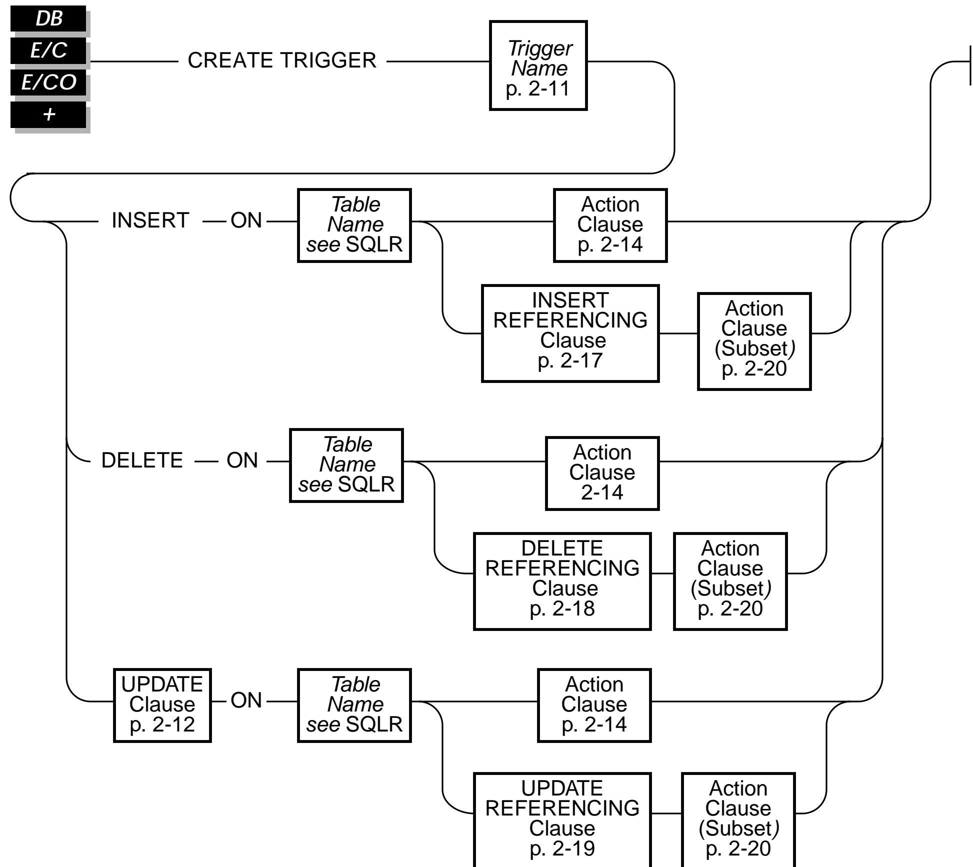
- CREATE TRIGGER
- DROP TRIGGER

CREATE TRIGGER

Purpose

Use the CREATE TRIGGER statement to create a new trigger.

Syntax



Usage

You must be either the owner of the table or the database administrator (DBA) to create a trigger on a table.

You can define a trigger with a standalone CREATE TRIGGER statement, or you can define it as part of a schema by placing the CREATE TRIGGER statement inside a CREATE SCHEMA statement.

You can only create a trigger on a table in the current database. You cannot create a trigger on a temporary table, a view, or a system catalog table.

You cannot create a trigger inside a stored procedure if the procedure is called inside a data manipulation statement. For example, you cannot create a trigger inside the stored procedure **sp_items** in the following INSERT statement:

```
INSERT INTO items EXECUTE PROCEDURE sp_items
```

See Chapter 7 of *The Informix Guide to SQL: Reference* for a list of data manipulation statements.

For each table, you can define only one trigger that is activated by an INSERT statement and only one trigger that is activated by a DELETE statement. For each table, you can define multiple triggers that are activated by UPDATE statements. See "UPDATE Clause" on page 2-12 for more information about multiple triggers on the same table.

E/C
E/CO

If you are embedding the CREATE TRIGGER statement in an **ESQL/C** or **ESQL/COBOL** program, you cannot use a host variable in the trigger specification.

You cannot use a stored procedure variable in a CREATE TRIGGER statement.

The Trigger Event

The trigger event is the INSERT, DELETE, or UPDATE statement that activates the trigger. Each trigger can have only one trigger event.

It is your responsibility to guarantee that the triggering statement returns the same result with and without the triggered actions. See "Action Clause" on page 2-14 and "Triggered Action" on page 2-21 for more information on the behavior of triggered actions.

STAR

If **INFORMIX-OnLine** is the database server, a triggering statement from an external database server can activate the trigger. As shown in the following example, an insert trigger on **newtab**, managed by **dbserver1**, is set off by an INSERT statement from **dbserver2**. The trigger executes just as if the insert originated on **dbserver1**.

```
-- Trigger on stores5@dbserver1:newtab

CREATE TRIGGER ins_tr INSERT ON newtab
REFERENCING new AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE nt_pct (post_ins.mc));

-- Triggering statement from dbserver2

INSERT INTO stores5@dbserver1:newtab
    SELECT item_num, order_num, quantity, stock_num, manu_code,
    total_price FROM items;
```

An insert trigger set off by an insert to an external database

Trigger Events with Cursors

If the triggering statement uses a cursor, the complete trigger is activated once for each execution of the statement. For example, if you declare a cursor for a triggering INSERT statement, each PUT statement executes the complete trigger. Similarly, if a triggering UPDATE or DELETE statement contains the clause WHERE CURRENT OF, each update or delete activates the complete trigger. Note that this behavior is different from what occurs when a triggering statement does not use a cursor and updates multiple rows. In this case, the set of triggered actions is executed only once. See “Action Clause” on page 2-14 for more information on the execution of triggered actions.

Privileges on the Trigger Event

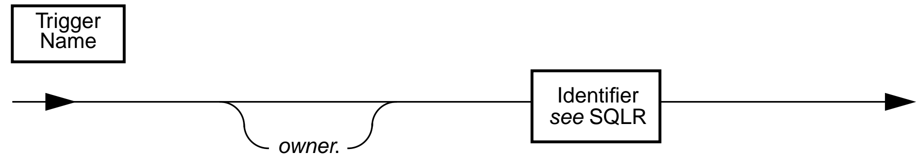
You must have the appropriate Insert, Delete, or Update privilege on the triggering table to execute the INSERT, DELETE, or UPDATE statement that is the trigger event. The triggering statement might still fail, however, if you do not have the privileges necessary to execute one of the SQL statements in the action clause. When the triggered actions are executed, the database server checks your privileges for each SQL statement in the trigger definition as if the statement were being executed independently of the trigger. See “Privileges to Execute Triggered Actions” on page 2-28 for information on the privileges you need to execute a trigger.

Impact of Triggers

The INSERT, DELETE, and UPDATE statements that initiate triggers might appear to execute slowly because they activate additional SQL statements, and the user might not know that other actions are occurring.

The execution time for a triggering data manipulation statement depends on the complexity of the triggered action and whether it, in turn, initiates other triggers. Obviously, the elapsed time for the triggering data manipulation statement increases as the number of cascading triggers increases. See "Cascading Triggers" on page 2-29 for more information on triggers initiating other triggers.

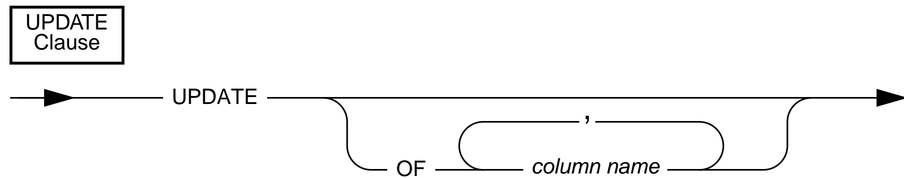
Trigger Name



owner is the user name of the owner of the trigger.

The trigger name follows the same syntax rules as other SQL identifiers. See “Identifier” in Chapter 7 of *The Informix Guide to SQL: Reference.*

UPDATE Clause



column name is the name of a column that will activate the trigger.

If the trigger event is an UPDATE statement, the trigger executes if any of the columns in the triggering column list are updated.

If you specify one or more triggering column names, the column names must belong to the table on which you create the trigger. If you do not specify a list of triggering columns, the default list consists of all the columns in the table on which you create the trigger.

If the triggering UPDATE statement updates more than one of the triggering columns in a trigger, the trigger only executes once.

Defining Multiple Update Triggers

If you define more than one update trigger event on a table, the column lists of the triggers must be mutually exclusive. For example, of the following triggers on the **items** table, **trig3** is illegal because its column list includes **stock_num**, which is a triggering column in **trig1**.

```
CREATE TRIGGER trig1 UPDATE OF item_num, stock_num ON items
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW(EXECUTE PROCEDURE proc1);

CREATE TRIGGER trig2 UPDATE OF manu_code ON items
BEFORE(EXECUTE PROCEDURE proc2);

-- Illegal trigger: stock_num occurs in trig1
CREATE TRIGGER trig3 UPDATE OF order_num, stock_num ON items
BEFORE(EXECUTE PROCEDURE proc3);
```

Multiple update triggers on a table* cannot *include the same columns

When an UPDATE Statement Activates Multiple Triggers

When an UPDATE statement updates multiple columns that have different triggers, the column numbers of the triggering columns determine the order of trigger execution. Execution begins with the smallest triggering column number and proceeds in order to the largest triggering column number. For example, table **taba** has four columns (**a, b, c, d**), as follows:

```
CREATE TABLE taba (a int, b int, c int, d int)
```

If you define **trig1** as an update on columns **a** and **c**, and **trig2** as an update on columns **b** and **d,** as follows:

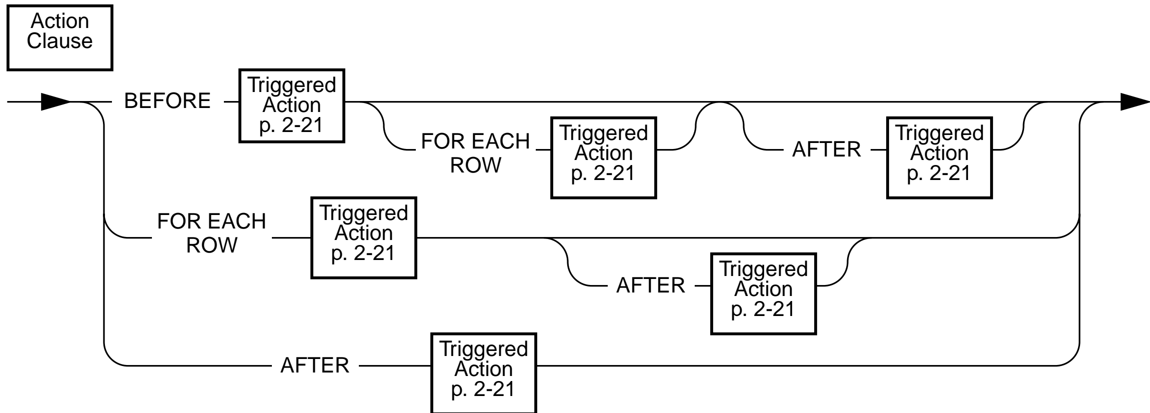
```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
    AFTER (UPDATE tabb SET y = y + 1);

CREATE TRIGGER trig2 UPDATE OF b, d ON taba
    AFTER (UPDATE tabb SET z = z + 1);
```

and the triggering statement is

```
UPDATE taba SET (b, c) = (b + 1, c + 1)
```

then **trig1** for columns **a** and **c** executes first, and **trig2** for columns **b** and **d** executes next. In this case, the smallest column number in the two triggers is column 1 (**a**) and the next is column 2 (**b**).

Action Clause



The action clause defines the triggered actions and specifies when they occur. You must define at least one triggered action, using the keywords BEFORE, FOR EACH ROW, or AFTER to indicate when the action is to occur, relative to the triggering statement. You can define triggered actions for all three options on a single trigger, but you must order them in sequence: BEFORE, FOR EACH ROW, and then AFTER. You cannot follow a FOR EACH ROW triggered action list with a BEFORE triggered action list. If the first triggered action list is FOR EACH ROW, an AFTER action list is the only option that can follow it. See "Action Clause Subset" on page 2-20 for more information on the action clause when a REFERENCING clause is present.

BEFORE Actions

The BEFORE triggered action executes once before the triggering statement executes. If the triggering statement does not process any rows, the BEFORE triggered actions still execute because it is not yet known whether any row is affected.

FOR EACH ROW Actions

The FOR EACH ROW triggered action executes once for each row that the triggering statement affects. The triggered SQL statement executes after the triggering statement processes each row.

If the triggering statement does not insert, delete, or update any rows, the FOR EACH ROW triggered actions do not execute.

AFTER Actions

An AFTER triggered action executes once after the action of the triggering statement is complete. If the triggering statement does not process any rows, the AFTER triggered action still executes.

Actions of Multiple Triggers

When an UPDATE statement activates multiple triggers, the triggered actions are merged. For example, assume that **taba** has columns **a**, **b**, **c**, and **d** as follows:

```
CREATE TABLE taba (a int, b int, c int, d int)
```

Next, assume that you define **trig1** on columns **a** and **c**, and **trig2** on columns **b** and **d**. If both triggers have triggered actions that are executed BEFORE, FOR EACH ROW, and AFTER, then the triggered actions are executed in the following sequence:

1. BEFORE action list for trigger (a, c)
2. BEFORE action list for trigger (b, d)
3. FOR EACH ROW action list for trigger (a, c)
4. FOR EACH ROW action list for trigger (b, d)
5. AFTER action list for trigger (a, c)
6. AFTER action list for trigger (b, d)

The database server treats the triggers as a single trigger, and the triggered action is the merged action list. All the rules governing a triggered action apply to the merged list as one list, and no distinction is made between the two original triggers.

Guaranteeing Row-Order Independence

When a triggered action specifies FOR EACH ROW, the result might depend on the order of the rows being processed. You can ensure that the result is independent of row order by *avoiding* the following actions.

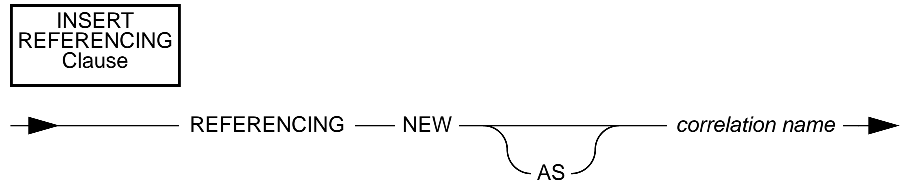
- Selecting the triggering table in the FOR EACH ROW section. If the triggering statement affects multiple rows in the triggering table, the result of the

SELECT statement in the FOR EACH ROW section varies as each row is processed. This also applies for any cascading triggers. (See "Cascading Triggers" on page 2-29.)

- In the FOR EACH ROW section, updating a table with values derived from the current row of the triggering table. If the triggered actions modify any row in the table more than once, the final result for that row depends on the order in which rows from the triggering table are processed.
- Modifying a table in the FOR EACH ROW section that is selected by another triggered statement in the same FOR EACH ROW section, including any cascading triggered actions. If you modify a table in this section and later refer to it, the changes to the table might not be complete at the time you refer to it. Consequently, the result might differ depending on the order in which rows are processed.

The database server does not enforce rules to prevent these situations because doing so would restrict the set of tables from which a triggered action can select. Furthermore, the result of most triggered actions is independent of row order. Consequently, you are responsible for ensuring that the results of the triggered actions are independent of row order.

INSERT REFERENCING Clause



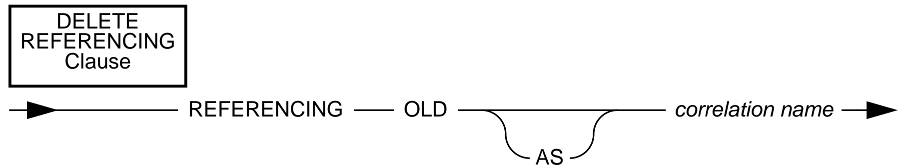
correlation name is a name you assign to a new column value so that you can refer to it within the triggered actions. The new column value is the column value after the triggering statement executes. Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. (See “Action Clause Subset” on page 2-20.) The correlation name follows the same syntax rules as other identifiers. (See “Identifier” in Chapter 7 of *The Informix Guide to SQL: Reference*.) The correlation name must be unique within the CREATE TRIGGER statement.

To use the correlation name, precede the column name with the correlation name, followed by a period. For example, if the new correlation name is **post**, you refer to the new value for the column **fname** as **post.fname**.

If the trigger event is an INSERT statement, use of the old correlation name as a qualifier causes an error because no value exists before the row is inserted. See “Using Correlation Names in Triggered Actions” on page 2-24 for the rules governing the use of correlation names.

You can use the INSERT REFERENCING clause only if you define a FOR EACH ROW triggered action.

DELETE REFERENCING Clause



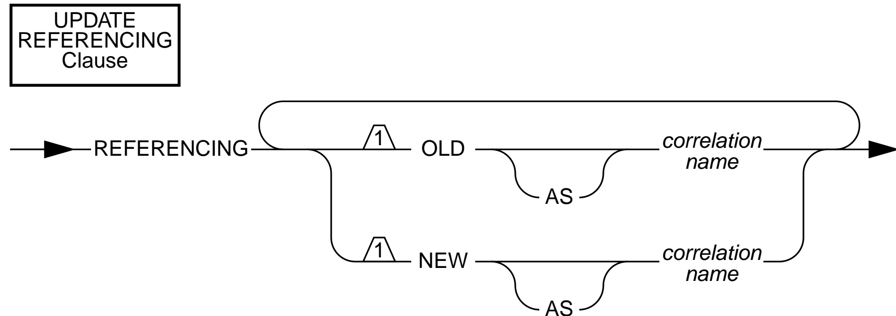
correlation name is a name you assign to an old column value so that you can reference it within the triggered actions. The old column value in the triggering table is its value before the triggering statement executes. Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. (See "Action Clause Subset" on page 2-20.) The correlation name follows the same syntax rules as other identifiers. (See "Identifier" in Chapter 7 of *The Informix Guide to SQL: Reference*.) The correlation name must be unique within the CREATE TRIGGER statement.

You use the correlation name to refer to an old column value by preceding the column name with the correlation name and a period (.). For example, if the old correlation name is **pre**, you refer to the old value for the column **fname** as **pre.fname**.

If the trigger event is a DELETE statement, use of the new correlation name as a qualifier causes an error because the column has no value after the row is deleted. See "Using Correlation Names in Triggered Actions" on page 2-24 for the rules governing the use of correlation names.

You can use the DELETE REFERENCING clause only if you define a FOR EACH ROW triggered action.

UPDATE REFERENCING Clause



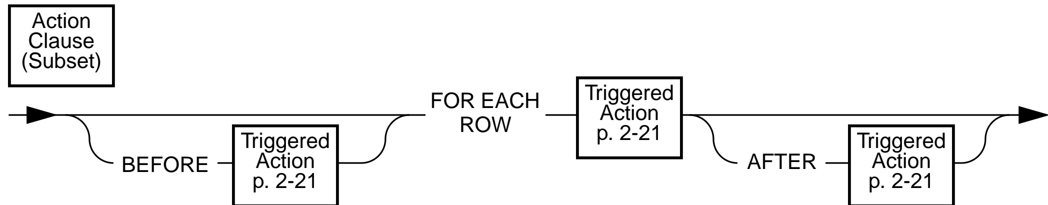
correlation name is a name you assign to an old or new column value so that you can refer to that value within the triggered action. The old column value in the triggering table is its value before the triggering statement made the change; its new value is its value after the triggering statement executes. Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. (See "Action Clause Subset" on page 2-20.) The correlation name follows the same syntax rules as other identifiers. (See "Identifier" in Chapter 7 of *The Informix Guide to SQL: Reference*.) The correlation name must be unique within the CREATE TRIGGER statement.

You use the correlation name to refer to an old or new column value by preceding the column name with the correlation name and a period (.). For example, if the new correlation name is **post**, you refer to the new value for the column **fname** as **post.fname**.

If the trigger event is an UPDATE statement, you can define both old and new correlation names to refer to column values before and after the triggering update. See "Using Correlation Names in Triggered Actions" on page 2-24 for the rules governing the use of correlation names.

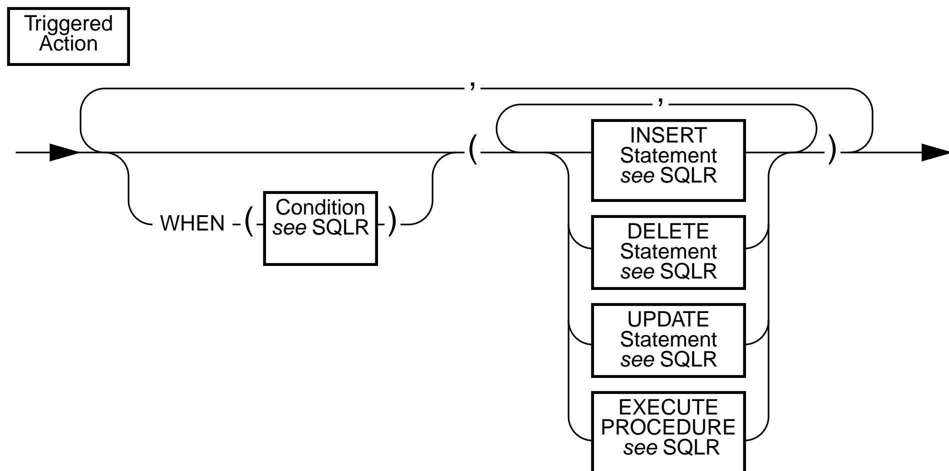
You can use the UPDATE REFERENCING clause only if you define a FOR EACH ROW triggered action.

Action Clause Subset



If the CREATE TRIGGER statement contains an INSERT REFERENCING clause, a DELETE REFERENCING clause, or an UPDATE REFERENCING clause, you *must* include a FOR EACH ROW triggered action section in the action clause. You can also include BEFORE and AFTER triggered action sections, but they are optional. See "Action Clause" on page 2-14 for information on the BEFORE, FOR EACH ROW, and AFTER triggered action sections.

Triggered Action



The triggered action consists of an optional WHEN condition and the action statements. Objects that are referenced in the triggered action—that is, tables, columns, and stored procedures—must exist when the CREATE TRIGGER statement is executed. This applies only to objects that are directly referenced in the trigger definition.

The WHEN Condition

The WHEN condition lets you make the triggered action dependent on the outcome of a test. When you include a WHEN condition in a triggered action, if the triggered action evaluates to *true*, the actions in the triggered action list execute in the order in which they appear. If the WHEN condition evaluates to *false* or *unknown*, the actions in the triggered action list are not executed. If the triggered action specifies FOR EACH ROW, its search condition is evaluated for each row also.

For example, the triggered action in the following trigger only executes if the condition in the WHEN clause is true:

```
CREATE TRIGGER up_price
UPDATE OF unit_price ON stock
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
    (INSERT INTO warn_tab VALUES(pre.stock_num, pre.order_num,
        pre.unit_price, post.unit_price, CURRENT))
```

Triggered action with optional WHEN condition

A stored procedure that executes inside the WHEN condition carries the same restrictions as a stored procedure that is called in a data manipulation statement. See the CREATE PROCEDURE statement in Chapter 7 of *The Informix Guide to SQL: Reference* for more information about a stored procedure that is called within a data manipulation statement.

The Action Statements

The triggered action statements can be INSERT, DELETE, UPDATE, or EXECUTE PROCEDURE statements. If a triggered action list contains multiple statements, these statements are executed in the order in which they appear in the list.

SE

In **INFORMIX-SE**, all objects referenced in the triggered actions must be in the current database.

Achieving a Consistent Result

To guarantee that the triggering statement returns the same result with and without the triggered actions, make sure that the triggered actions in the BEFORE and FOR EACH ROW sections do not modify any table referenced in the following clauses:

- WHERE clause
- SET clause in the UPDATE statement
- SELECT clause
- EXECUTE PROCEDURE clause in a multiple row INSERT statement

Using Keywords

If you use the keywords INSERT, DELETE, UPDATE, or EXECUTE as an identifier in any of the following clauses inside the triggered action, you must qualify them by the owner name, or the table name, or both:

- FROM clause of a SELECT statement
- INTO clause of the EXECUTE PROCEDURE statement
- GROUP BY clause
- SET clause of the UPDATE statement

A syntax error is returned if these keywords are *not* qualified when used in these clauses inside a triggered action.

If you use the keyword as a column name, it must be qualified by the table name—for example, **table.update**. If both the table name and the column name are keywords, they must be qualified by the owner name—for example, **owner.insert.update**. If the owner name, table name, and column name are all keywords, the owner name must be in quotes—for example, **"delete".insert.update**. The only exception is when these keywords are the first table or column name in the list. In that case, you do not have to qualify them. For example, **delete** in the following statement does *not* need to be qualified because it is the first column listed in the INTO clause.

```
CREATE TRIGGER t1 UPDATE OF b ON tab1
    FOR EACH ROW (EXECUTE PROCEDURE p2()
    INTO delete, d)
```

The following examples show instances where you must qualify the column name or the table name:

```
CREATE TRIGGER t1 INSERT ON tab1
    BEFORE (INSERT INTO tab2 SELECT * FROM tab3,
        "owner1".update)
```

FROM clause of a SELECT statement

```
CREATE TRIGGER t3 UPDATE OF b ON tab1
    FOR EACH ROW (EXECUTE PROCEDURE p2() INTO
        d, tab1.delete)
```

INTO clause of the EXECUTE PROCEDURE statement

```
CREATE TRIGGER t4 DELETE ON tab1
    BEFORE (INSERT INTO tab3 SELECT deptno, SUM(exp)
    FROM budget GROUP BY deptno, budget.update)
```

GROUP BY clause

```
CREATE TRIGGER t2 UPDATE OF a ON tab1
    BEFORE (UPDATE tab2 SET a = 10, tab2.insert = 5)
```

SET clause of the UPDATE statement

Using Correlation Names in Triggered Actions

The following rules apply when you use correlation names in triggered actions:

- You can use the correlation names for the old and new column values only in statements in the FOR EACH ROW triggered action section. You can use the old and new correlation names to qualify any column in the triggering table in either the WHEN condition or the triggered SQL statements.
- The old and new correlation names refer to all rows affected by the triggering statement.
- You cannot use the correlation name to qualify a column name in the GROUP BY clause, the SET clause, or the COUNT DISTINCT clause.
- The scope of the correlation names for the old and new column values is the entire trigger definition. This scope is statically determined, meaning that it is limited to the trigger definition. Thus, it does not encompass cascading triggers or columns that are qualified by a table name in a stored procedure that is a triggered action.

When to Use Correlation Names

In an SQL statement in a FOR EACH ROW triggered action, you must qualify all references to columns in the triggering table with either the old or new correlation name, unless the statement is valid independent of the triggered action.

In other words, if a column name inside a FOR EACH ROW triggered action section is not qualified by a correlation name, even if it is qualified by the triggering table name, it is interpreted as if the statement were independent of the triggered action. No special effort is made to search the definition of the triggering table for the nonqualified column name.

For example, assume that the following DELETE statement is a triggered action inside the FOR EACH ROW section of a trigger.

```
DELETE FROM tab1 WHERE col_c = col_c2
```

For the statement to be valid , both **col_c** and **col_c2** must be columns from **tab1**. If **col_c2** is intended to be a correlation reference to a column in the triggering table, it must be qualified by either the old or the new correlation name. If **col_c2** is not a column in **tab1** and it is not qualified by either the old or new correlation name, an error is returned.

When a column is not qualified by a correlation name, and the statement is valid independent of the triggered action, the column name refers to the current value in the database. In the triggered action for trigger **t1** in the following example, **mgr** in the WHERE clause of the correlated subquery is an unqualified column from the triggering table. In this case, **mgr** refers to the current column value in **empsal** because the INSERT statement is valid independent of the triggered action.

```
CREATE DATABASE db1;
CREATE TABLE empsal (empno INT, salary INT, mgr INT);
CREATE TABLE mgr (eno INT, bonus INT);
CREATE TABLE biggap (empno INT, salary INT, mgr INT);

CREATE TRIGGER t1 UPDATE OF salary ON empsal
AFTER (INSERT INTO biggap SELECT * FROM empsal WHERE salary <
        (SELECT bonus FROM mgr WHERE eno = mgr));
```

In a triggered action, an unqualified column name from the triggering table refers to the current column value, but only when the triggered statement is valid independent of the triggered action.

Qualified Versus Unqualified Value

The following table summarizes the value retrieved when you use the column name qualified by the old correlation name, and the column name qualified by the new correlation name.

Trigger Event	**old.col**	**new.col**
INSERT	no value (error)	inserted value
UPDATE (column updated)	original value	current value (N)
UPDATE (column not updated)	original value	current value (U)
DELETE	original value	no value (error)

Refer to this key when reading the table:

original	value before the triggering statement
current	value after the triggering statement
N	cannot be changed by triggered action
U	can be updated by triggered statements; value may be different than original value because of preceding triggered actions

Outside the FOR EACH ROW section, you cannot qualify a column from the triggering table with either the old correlation name or the new correlation name; thus, it always refers to the current value in the database.

Action on the Triggering Table

You cannot reference the triggering table in any triggered SQL statement, with the following two exceptions:

- The trigger event is UPDATE and the triggered SQL statement is also UPDATE, and the columns in both statements, including any nontriggering columns in the triggering UPDATE, are mutually exclusive.

  For example, if the following UPDATE statement, which updates columns **a** and **b** of **tab1**, is the triggering statement:

  ```
  UPDATE tab1 SET (a, b) = (a + 1, b + 1)
  ```

  then the first UPDATE statement in the following pair is a valid triggered action, but the second one is not because it updates column **b** again:

  ```
  UPDATE tab1 SET c = c + 1; -- OK
  UPDATE tab1 SET b = b + 1;-- ILLEGAL
  ```

- The triggered SQL statement is a SELECT statement. The SELECT statement can be a triggered statement in three instances:
  - The SELECT statement appears in a subquery in the WHEN clause.
  - The triggered action is a stored procedure and the SELECT statement appears inside the stored procedure.
  - The SELECT statement appears in any subquery, such as an INSERT with a SELECT statement as a subquery.

This rule, which states that a triggered SQL statement cannot reference the triggering table, with the two noted exceptions, applies recursively to all cascading triggers, which are considered part of the initial trigger. This means that a cascading trigger cannot update any of the columns in the triggering table that were updated by the original triggering statement, including any nontriggering columns affected by that statement. For example, if the following UPDATE statement is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1)
```

then in the cascading triggers that follow, **trig2** will fail at run time because it references column **b**, which is updated by the triggering UPDATE statement. See "Cascading Triggers" on page 2-29 for more information about cascading triggers.

```
CREATE TRIGGER trig1 UPDATE OF a ON tab1    -- Valid
    AFTER (UPDATE tab2 set e = e + 1);

CREATE TRIGGER trig2 UPDATE of e ON tab2    -- Invalid
    AFTER (UPDATE tab1 set b = b + 1);
```

Rules for Stored Procedures

The following rules apply to a stored procedure that is used as a triggered action:

- The stored procedure cannot be a cursory procedure (that is, a procedure that returns more than one row) in a place where only one row is expected.
- When an EXECUTE PROCEDURE statement is the triggered action, you can specify the INTO clause only for an UPDATE trigger when the triggered action occurs in the FOR EACH ROW section. In this case, the INTO clause can contain only column names from the triggering table. The following statement illustrates the appropriate use of the INTO clause:

  ```
  CREATE TRIGGER upd_totpr UPDATE OF quantity ON items
  REFERENCING OLD AS pre_upd NEW AS post_upd
  FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
      post_upd.quantity, pre_upd.total_price)
      INTO total_price)
  ```

  When the INTO clause appears in the EXECUTE PROCEDURE statement, the database server updates the columns named there with the values returned from the stored procedure. The database server performs the update immediately upon returning from the stored procedure. See the EXECUTE PROCEDURE statement in Chapter 7 of *The Informix Guide to SQL: Reference* for more information about the statement.
- You cannot use the old correlation name and the new correlation name inside the stored procedure. If you need to use the corresponding values in the procedure, you must pass them as parameters. The reason for this is that the stored procedure should be independent of triggers, and the

new correlation name and the old correlation name do not have any meaning outside the trigger.

- You cannot use a BEGIN WORK, COMMIT WORK, ROLLBACK WORK, or SET CONSTRAINTS statement.

When you use a stored procedure as a triggered action, the objects that it references are not checked until the procedure is executed.

Privileges to Execute Triggered Actions

If you are not the owner of the trigger, and if the owner's privileges include the WITH GRANT OPTION right, you inherit the owner's privileges, including the WITH GRANT OPTION right, for each triggered SQL statement. These privileges are in addition to your own privileges.

If the triggered action is a stored procedure, you must have Execute privilege on the procedure, or the owner of the trigger must have Execute privilege and the WITH GRANT OPTION right. Inside the stored procedure, however, you do not carry the privileges of the owner of the trigger. Here you carry the privileges of the owner of the procedure, or the privileges of a DBA, if it is a DBA-privileged procedure.

For an owner-privileged procedure, if the procedure owner has the WITH GRANT OPTION right, you inherit the owner's privileges. In this case, all of the nonqualified objects referenced in the procedure are qualified by the name of the owner of the procedure.

For a DBA-privileged procedure, you have the privileges of the DBA. In this case, the non-qualified objects referenced in the procedure are qualified by your user name. See Chapter 8 of *The Informix Guide to SQL: Reference* for more information on privileges on stored procedures.

Creating a Triggered Action Anyone Can Use

To create a trigger that is executable by anyone who has privileges to execute the triggering statement, you can ask the DBA to create a DBA-privileged procedure and grant you the Execute privilege with the WITH GRANT OPTION right. You then use the DBA-privileged procedure as the triggered action. Anyone can execute the triggered action because the the DBA-privileged procedure carries the WITH GRANT OPTION right. When you activate the procedure, the database server applies privilege-checking rules for a DBA. See Chapter 8 of *The Informix Guide to SQL: Reference* for more information on privileges on stored procedures.

Cascading Triggers

The database server allows triggers to cascade, meaning that the triggered actions of a trigger can activate another trigger. The maximum number of triggers in a cascading sequence is 61, the initial trigger plus a maximum of 60 cascading triggers. When the the number of cascading triggers in a series exceeds the maximum, the database server returns error number -748:

```
Exceeded limit on maximum number of cascaded triggers.
```

The following example illustrates a series of cascading triggers that enforce referential integrity on the **manufact**, **stock**, and **items** tables in the **stores5** database. When a manufacturer is deleted from the **manufact** table, the first trigger, **del_manu**, deletes all the manufacturer's items from the **stock** table. Each delete in the **stock** table activates a second trigger, **del_items**, that deletes all the manufacturer's **items** from the **items** table. Finally, each delete in the **items** table triggers the stored procedure **log_order**, which creates a record of any orders in the **orders** table that can no longer be filled.

```
CREATE TRIGGER del_manu
DELETE ON manufact
REFERENCING OLD AS pre_del
FOR EACH ROW(DELETE FROM stock
    WHERE manu_code = pre_del.manu_code);

CREATE TRIGGER del_stock
DELETE ON stock
REFERENCING OLD AS pre_del
FOR EACH ROW(DELETE FROM items
    WHERE manu_code = pre_del.manu_code);

CREATE TRIGGER del_items
DELETE ON items
REFERENCING OLD AS pre_del
FOR EACH ROW(EXECUTE PROCEDURE log_order(pre_del.order_num));
```

Cascading triggers

Note that when you are using either the **INFORMIX-SE** database server, or the **INFORMIX-OnLine** database server without logging, primary key constraints on both the **manufact** and **stock** tables would prohibit the triggers in this example. When you use **INFORMIX-OnLine** with logging, the triggers execute successfully because constraint checking is deferred until the triggered actions are complete, including the actions of cascading triggers. See "Constraint Checking" on page 2-30 for more information about how constraints are handled when triggers execute.

The database server prevents endless loops of cascading triggers by not allowing you to modify the triggering table in any of the cascading triggered actions, with the exception of an UPDATE statement that *does not* modify any of the columns updated by the triggering UPDATE statement.

Constraint Checking

For an **INFORMIX-OnLine** database with logging, **OnLine** defers constraint checking on the triggering statement until the statements in the triggered action list execute. **OnLine** effectively executes a SET CONSTRAINTS ALL DEFERRED statement before executing the triggering statement. At the completion of the triggered action, it effectively executes a SET CONSTRAINTS *constr_name* IMMEDIATE statement to immediately check the constraints that were deferred. This allows you to write triggers in such a way that the triggered action can resolve any constraint violations that the triggering statement creates.

Consider the following example, in which the table **child** has constraint **r1** that references the table **parent**. You define trigger **trig1** and activate it with an INSERT statement. In the triggered action, **trig1** checks to see if **parent** has a row with the value of the current **cola** in **child**. If not, it inserts it.

```
CREATE TABLE parent (cola INT PRIMARY KEY);
CREATE TABLE child (cola INT REFERENCES parent CONSTRAINT r1);
CREATE TRIGGER trig1 INSERT ON child
    REFERENCING NEW AS new
    FOR EACH ROW
    WHEN((SELECT COUNT (*) FROM parent
        WHERE cola = new.cola) = 0)
-- parent row does not exist
    (INSERT INTO parent VALUES (new.cola));
```

When you insert a row to a table that is the child table in a referential constraint, the row might not exist in the parent table. The database server does not immediately return this error on a triggering statement. Instead, it allows the triggered action to resolve the constraint violation by inserting the corresponding row into the parent table. As shown in the example, within the triggered action you can check whether the parent row exists and, if so, bypass the insert.

For an **INFORMIX-OnLine** database without logging, **OnLine** does *not* defer constraint checking on the triggering statement. In this case, it immediately returns an error if the triggering statement violates a constraint.

INFORMIX-OnLine does not allow the SET CONSTRAINTS statement in a triggered action. **OnLine** checks this restriction when you activate a trigger because the statement could occur inside a stored procedure.

SE

For an **INFORMIX-SE** database, with or without logging, constraint checking occurs prior to the triggered action. If a constraint violation results from the triggering statement, **INFORMIX-SE** returns an error immediately.

Preventing Triggers from Overriding Each Other

When you activate multiple triggers with an UPDATE statement, it is possible for a trigger to override the changes made by an earlier trigger. If you do not want the triggered actions to interact, you can split the UPDATE statement into multiple UPDATE statements, each of which updates an individual column. As another alternative, you can create a single update trigger for all columns that require triggered action. Then, inside the triggered action, you can test for the column being updated and apply the actions in the desired order. This approach, however, is different than having the database server apply the actions of individual triggers, and it has the following disadvantages:

- If the trigger has a BEFORE action, it applies to all columns because you cannot yet detect whether a column has changed.
- If the triggering UPDATE statement sets a column to the same value it holds, you cannot detect the update and, therefore, the triggered action is skipped. You might want to execute the triggered action even though the value of the column is not changed.

The Client/Server Environment

STAR

In a database under **INFORMIX-OnLine**, the statements inside the triggered action can affect tables in external databases. The following example shows an update trigger on **dbserver1** triggering an update to **items** on **dbserver2**.

```
CREATE TRIGGER upd_nt UPDATE ON newtab
REFERENCING new AS post
FOR EACH ROW(UPDATE stores5@dbserver2:items
  SET quantity = post.qty WHERE stock_num = post.stock
  AND manu_code = post.mc)
```

A triggered action affecting a table in an external database (*INFORMIX-OnLine *only)

If a statement from an external database server initiates the trigger, however, and the triggered action affects tables in an external database, the triggered actions fail. For example, the following combination of triggered action and triggering statement results in an error when the triggering statement executes:

```
-- Triggered action from dbserver1 to dbserver3:

CREATE TRIGGER upd_nt UPDATE ON newtab
REFERENCING new AS post
FOR EACH ROW(UPDATE stores5@dbserver3:items
     SET quantity = post.qty WHERE stock_num = post.stock
     AND manu_code = post.mc);

-- Triggering statement from dbserver2:

UPDATE stores5@dbserver1:newtab
     SET qty = qty * 2 WHERE s_num = 5
     AND mc = "ANZ";
```

Example of an external triggering statement and an external triggered action that fail

INET

In a database under **INFORMIX-SE**, all objects referenced in the triggered actions must be in the current database.

Logging and Recovery

You can create triggers for databases both with and without logging. However, when the database does not have logging, there is no rollback when the triggering statement fails. In this case, it is your responsibility to maintain data integrity in the database.

In **INFORMIX-OnLine**, if the trigger fails and the database has transactions, all triggered actions and the triggering statement are rolled back because the triggered actions are an extension of the triggering statement. The rest of the transaction, however, is not rolled back.

SE

In **INFORMIX-SE**, if you explicitly begin a transaction, you must explicitly roll back the whole transaction. If the database has no transactions, there is a possibility that data integrity might be violated when the triggered actions fail.

For **INFORMIX-SE**, even if the database has logging, any data definition statement in the triggered action cannot be rolled back. Again, it is your responsibility to maintain data integrity and integrity of the database structure.

Note that the row action of the triggering statement occurs before the triggered actions in the FOR EACH ROW section. If the triggered action fails for a database without logging, the application must restore the row that was changed by the triggering statement to its previous value.

When you use a stored procedure as a triggered action, if you terminate the procedure in an exception-handling section, any actions that modify data inside that section are rolled back along with the triggering statement. For example, in the following excerpt, when the exception handler traps an error it inserts a row into the **logtab** table.

```
ON EXCEPTION IN (-201)
    INSERT INTO logtab values (errno, errstr);
    RAISE EXCEPTION -201
END EXCEPTION
```

When the RAISE EXCEPTION statement returns the error, however, the database server rolls back this insert because it is part of the triggered actions. If the procedure is executed outside a triggered action, the insert is not rolled back.

The stored procedure that implements a triggered action cannot contain any BEGIN WORK, COMMIT WORK, or ROLLBACK WORK statements. If the database has logging, you must either begin an explicit transaction before the triggering statement, or the statement itself must be an implicit transaction. In any case, another transaction-related statement cannot appear inside the stored procedure.

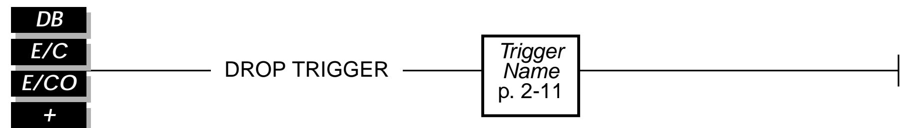
You can use triggers to enforce referential actions that the database server does not currently support. Again, however, for an **INFORMIX-SE** database or for an **INFORMIX-OnLine** database without logging, you are responsible for maintaining data integrity when the triggering statement fails.

DROP TRIGGER

Purpose

Use the DROP TRIGGER statement to drop a trigger definition from a table.

Syntax



Usage

You must be the owner of the trigger or the DBA to drop a trigger.

The following statement drops the **items_pct** trigger:

```
DROP TRIGGER items_pct
```

You cannot drop a trigger inside a stored procedure if the procedure is called within a data manipulation statement. For example, in the following INSERT statement, a DROP TRIGGER statement is illegal inside the stored procedure **proc1**:

```
INSERT INTO orders EXECUTE PROCEDURE proc1(vala, valb)
```

See the CREATE PROCEDURE statement in Chapter 7 of *The Informix Guide to SQL: Reference* for more information about a stored procedure that is called within a data manipulation statement.

Triggers and Other SQL Statements

The following SQL statements have implications for triggers. See Chapter 7 of *The Informix Guide to SQL: Reference* for more information about these statements.

ALTER TABLE	When you drop a column from a table with the ALTER TABLE statement, the column is dropped from the triggering column lists of all triggers defined for the table. If the column you drop is the only triggering column in a trigger, the trigger is dropped. If you just modify the definition of a triggering column with the ALTER TABLE statement, the trigger remains unchanged because it is assumed to still be valid.
CREATE SCHEMA	You can include the CREATE TRIGGER statement within the CREATE SCHEMA statement.
DROP DATABASE	The DROP DATABASE statement drops all triggers within the database.
DROP TABLE	The DROP TABLE statement drops all triggers for a table.
PREPARE	If a trigger is created or dropped after you prepare a triggering data manipulation statement for execution with the PREPARE statement, the prepared statement is invalid when you submit the EXECUTE statement for it.
RENAME TABLE	When you use the RENAME TABLE statement to rename a table that has a trigger, the database server replaces the name of the triggering table in the trigger definition. The old table name is not replaced, however, where it is referenced inside any triggered actions. If the new table name is the same as either the old correlation name or the new correlation name, an error is returned. When you activate a trigger, if the database server encounters an old table name in the triggered action, it returns an error when it cannot find the table.

RENAME COLUMN

When you use the RENAME COLUMN statement to rename a column that appears in a correlated reference inside the FOR EACH ROW section of a triggered action, the old column name is replaced with the new one in the triggered action. This extends to an EXECUTE PROCEDURE statement with an INTO clause, where the column name appears as a correlated reference in the INTO clause. The old column name is not replaced with the new one anywhere else in the triggered action lists.

If the column name appears in the UPDATE clause, it is replaced there with the new column name.

When you activate a trigger, if the database server encounters an old column name in the triggered action, it returns an error when it cannot find the column.

Appendix

A

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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

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

IBM Corporation
J74/G4
555 Bailey Ave
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

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

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

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

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

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

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

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

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

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

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

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

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

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

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Error Messages

This section contains a list of the Informix error messages that the database server can return when you create or use triggers. The list contains both new and changed messages. The messages are listed in ascending order with each error message followed by a corrective action.

-322 Cannot alter, rename, or create a trigger on view *view-name*.

You can only create a trigger on a table. Consider creating the trigger on the table from which the view is derived, or consider creating view *view-name* as a table and then creating the trigger on it.

-522 Table *table-name* not selected in query.

You used a correlation name to qualify a column name in either a GROUP BY clause or a SET clause. Consider rewriting the statement in a stored procedure that you then use as the triggered action, passing the column value as an argument. In any case, you must rewrite the statement without using a correlation name in the GROUP BY clause or the SET clause.

-542 Cannot specify a column more than once in a constraint or trigger.

You name the same column more than once in the triggering column list of an update trigger. Remove the duplicate occurrence of the column name and try again.

-548 No referential constraint or trigger allowed on a TEMP table.

You cannot create a constraint or a trigger on a temporary (TEMP) table. Consider creating the temporary table as a permanent table in the database. If this is a feasible option, create the table and then create the trigger on it.

-634 Object does not exist.

You are trying to drop a trigger that does not exist. Check that you are spelling the name of the trigger correctly. Also, you might query the **systriggers** system catalog table to review the names of triggers in the database.

-635 Not owner of object.

You are trying to drop a trigger that you do not own. You might query the **systriggers** system catalog table to see who owns the trigger. You probably need to ask the owner of the trigger or the DBA to drop the trigger.

-729 Trigger has no triggered action.

Your CREATE TRIGGER statement does not include a triggered action. Add a triggered action list to the trigger definition and then resubmit the CREATE TRIGGER statement.

-730 Cannot specify REFERENCING if trigger does not have FOR EACH ROW.

You included a REFERENCING clause in a CREATE TRIGGER statement that does not include a FOR EACH ROW triggered action section. Either remove the REFERENCING clause or, if it is appropriate, add the missing keywords FOR EACH ROW, followed by the triggered actions that are to occur at that time.

-731 Invalid use of column reference in trigger body.

For insert and delete triggers, this means the offending column is being used in the INTO clause of the EXECUTE PROCEDURE statement (which is only allowed for an update trigger). Remove the column names from the INTO clause.

-732 Incorrect use of old or new values correlation name inside trigger.

You cannot use the new or old correlation name outside the FOR EACH ROW section, or in the INTO clause of the EXECUTE PROCEDURE statement. It is also not valid to use the new or old correlation name to qualify the SELECT COUNT DISTINCT column. For example, the following statement returns error -732:

```
SELECT COUNT (DISTINCT oldname.colname)
```

You cannot specify an old correlation name for an insert trigger. You cannot specify a new correlation name for a delete trigger.

-733 Cannot reference procedure variable in CREATE TRIGGER statement.

You have a CREATE TRIGGER statement inside a stored procedure, and within the CREATE TRIGGER statement you reference a variable that is defined in the stored procedure. This is not legal. Remove the reference to the stored procedure variable from the CREATE TRIGGER statement, and try again.

-734 Object name matches old or new values correlation name.

This error is returned in three cases:

- The name of the triggering table, or the synonym, if it is used, matches the old or new correlation name in the REFERENCING clause.
- The name of a table or a synonym referenced in the action clause matches either the old or new correlation name in the REFERENCING clause.
- The old correlation name matches the new correlation name.

As appropriate, change either the correlation name or the table name and then execute the CREATE TRIGGER statement again.

-741 Trigger for the same event already exists.

You are creating a trigger for an event, and another trigger already exists for that event. You can only have one insert or delete trigger on a table. If you are defining multiple triggers that occur on an update, the column lists in the UPDATE statements must be mutually exclusive—that is, you cannot name a column as a triggering column in more than one UPDATE clause.

-743 Object *object_name* already exists in database.

You are trying to define an object that already exists in the database.

-744 Illegal SQL statement in trigger.

This error is returned when the triggered SQL statement is BEGIN WORK, COMMIT WORK, ROLLBACK WORK, or SET CONSTRAINTS. These statements are not allowed as triggered actions. Remove the offending statement.

-745 Trigger execution has failed.

This message is defined for general use to apply to error conditions that you specify in a stored procedure that is a triggered action.

-746 *message-string*

This is a message for which you supply *message-string*. You can apply this message to error conditions that you specify in a stored procedure that is a triggered action. The corrective action for this error depends on the condition that caused it. Both the condition and the message text are defined by you, the user.

-747 Table or column matches object referenced in triggering statement.

This error is returned when a triggered SQL statement acts on the triggering table, or when both statements are updates and the column being updated in the triggered action is the same as the column being updated by the triggering statement.

-748 Exceeded limit on maximum number of cascaded triggers.

You exceeded the maximum number of cascading triggers, which is 61. You may be setting off triggers without realizing it. You can query the **systriggers** system catalog table to find out what triggers exist in the database. You might consider tracing the triggered actions by placing the action clause of the initial trigger in a stored procedure and using the TRACE statement in SPL.

-749 Remote cursor operation disallowed with pre-5.01 server.

The triggering statement or cursor operation has been sent by an external pre-5.01 database server. This is not allowed.

-751 Remote procedure execution disallowed with pre-5.01 server.

The action clause of the trigger contains a stored procedure that is not called in a data manipulation statement, and the procedure will be executed by an external pre-5.01 database server. This is not allowed. A stored procedure that is called within a data manipulation statement is restricted from executing certain SQL statements, including transaction-related statements. Transaction-related statements are not allowed within a stored procedure that is a

triggered action. A pre-5.01 database server is not equipped to check for this, so the procedure is disallowed. If possible, execute the procedure on a 5.01 database server.

Index

Special symbols are listed in ASCII order at the end of the index.

A

Action clause
AFTER 2-15
consists of 1-5
definition of 2-14
example 1-5
FOR EACH ROW 2-14
keywords 1-5
syntax 2-14
Action clause subset, syntax 2-20
Action statements
in triggered action clause 2-22
list of 2-22
order of execution 2-22
Adding comments
ANSI-compliant method 1-6
non-ANSI-compliant method 1-6
to CREATE TRIGGER statement 1-6, 1-8
AFTER action 2-15
AFTER keyword 1-5, 2-15
ALTER TABLE statement 2-36
Applications for triggers 1-10

B

BEFORE keyword 1-5, 2-14
BEGIN WORK statement 2-28
Braces, adding comments with 1-6

C

Cascading triggers
- and triggering table 2-26, 2-30
- description of 2-29
- INFORMIX-OnLine 1-17
- INFORMIX-SE 1-17
- maximum number of 1-17, 2-29
- scope of correlation names 2-24
- triggered actions 2-16

Client/server environment 2-32

Column name
- in UPDATE clause 2-12
- when qualified 2-23

Column numbers, effect on triggers 2-13

Column value
- in triggered action 2-25
- qualified vs. unqualified 2-25
- when unqualified 2-25

Comments, including in CREATE TRIGGER statement 1-6

COMMIT WORK statement 2-28

Constraint checking 2-30

Conventions
- syntax Intro-5
- typographical Intro-5

Correlation name
- and stored procedures 2-24
- in COUNT DISTINCT clause 2-24
- in DELETE REFERENCING clause 2-18
- in GROUP BY clause 2-24
- in INSERT REFERENCING clause 2-17
- in SET clause 2-24
- in stored procedure 2-27
- in UPDATE REFERENCING clause 2-19
- new 2-19
- old 2-19
- rules for 2-24
- scope of 2-24
- table of values 2-25
- using 2-24
- when to use 2-24

COUNT DISTINCT clause 2-24

CREATE SCHEMA statement
- and CREATE TRIGGER statement 2-36
- defining a trigger 2-8

CREATE TRIGGER statement
- action clause 1-5
- adding comments 1-6, 1-8
- comments outside 1-7
- comments within 1-7
- elements of 1-4
- embedding in a program 1-7
- in ESQL/C 2-8
- in ESQL/COBOL 2-8
- privilege to use 2-8
- purpose 2-7
- REFERENCING clause 1-5
- syntax 2-7
- trigger event 1-4
- trigger name 1-4
- triggered action clause 2-21
- usage 2-8

Creating a trigger
- CREATE TRIGGER statement 1-4
- using DB-Access 1-6
- using ESQL/C 1-7
- using ESQL/COBOL 1-7

Creating an audit trail
- example 1-10
- using triggers 1-10

Cursor statement, as trigger event 2-9

D

Dashes, adding comments with 1-6

Database, stores5 Intro-13

datakey column, in systrigbody table 1-8

DB-Access
- creating a trigger with 1-6
- looking up a trigger header 1-9
- QUERY-LANGUAGE Menu 1-6

DELETE REFERENCING clause
- and FOR EACH ROW section 2-20
- correlation name 2-18
- syntax 2-18

DELETE statement
- as triggering statement 2-9
- in trigger event 2-8
- in triggered action 2-22

Demonstration database
- copying Intro-14
- installation script Intro-13
- overview Intro-13

Deriving data, use of triggers in 1-14

Documentation notes Intro-10

Documentation, other useful Intro-4
DROP DATABASE statement 2-36
DROP TABLE statement 2-36
DROP TRIGGER statement
syntax 2-35
use of 2-35

E

Embedding in a program
CREATE TRIGGER statement 1-7
in ESQL/C 1-7
in ESQL/COBOL 1-7
Enforcing referential integrity 1-16
Error messages
number -745 1-18
number -746 1-18
on SQL statements 1-18
retrieving text in a program 1-19, 1-21
text for error number -746 1-20
user-specified 1-18
using RAISE EXCEPTION statement 1-19
Example
creating an audit trail 1-10
deriving data 1-14
enforcing referential integrity 1-16
implementing a business rule 1-12
EXECUTE PROCEDURE statement, in triggered action 2-22

F

FOR EACH ROW action
SELECT statement in 2-16
triggered action list 2-14

G

GROUP BY clause 2-24

H

Header information
consists of 1-9
for a trigger 1-9

I

Icon, explanation of Intro-6
Informix products, application development tools Intro-3
INFORMIX-OnLine
and triggering statement 2-9
with logging 1-16
INFORMIX-SE, cascading triggers 1-17
INSERT REFERENCING clause
and FOR EACH ROW section 2-20
correlation name 2-17
syntax 2-17
INSERT statement
in trigger event 2-8
in triggered action 2-22

K

Keywords
AFTER 1-5, 2-14
BEFORE 1-5, 2-14
FOR EACH ROW 1-5, 2-14
NEW 2-18, 2-19
OLD 2-17, 2-18
using in triggered action 2-22

L

Looking up a trigger 1-8

M

Machine notes Intro-10
Multiple triggers
column numbers in 2-13
example 2-12
order of execution 2-13
preventing overriding 2-31

N

NEW keyword
in DELETE REFERENCING clause 2-18
in INSERT REFERENCING clause 2-17
in UPDATE REFERENCING clause 2-19

O

OLD keyword
in DELETE REFERENCING clause 2-18
in INSERT REFERENCING clause 2-17
in UPDATE REFERENCING clause 2-19
On-line
files Intro-10
help Intro-10
Order of execution, of action statements 2-22
Output from TRACE command 1-18

P

PREPARE statement 2-36
Privileges, for triggered action 2-28
PUT statement, impact on trigger 2-9

R

REFERENCING clause
defining 1-5
DELETE REFERENCING clause 2-18
INSERT REFERENCING clause 2-17
UPDATE REFERENCING clause 2-19
using referencing 2-24
Release notes Intro-10
RENAME COLUMN statement 2-36
RENAME TABLE statement 2-36
Result of triggering statement 2-22
ROLLBACK WORK statement 2-28
Row order, guaranteeing independence of 2-15
Rules for stored procedures 2-27

S

SELECT statement, in FOR EACH ROW section 2-16
SET clause 2-24
SET CONSTRAINT statement 2-28, 2-30
SPL. *See* Stored Procedure Language.
Stored procedure
as triggered action 2-27
checking references 2-28
DBA-privileged 2-28
in WHEN condition 2-22
owner-privileged 2-28
passing data to 1-15
privileges 2-28
tracing triggered actions 1-17
Stored Procedure Language (SPL)
SET DEBUG FILE TO 1-17
TRACE command 1-17
using trace commands 1-17
stores5 database
copying Intro-14
creating on INFORMIX-OnLine Intro-14
creating on INFORMIX-SE Intro-15
overview Intro-13
Syntax diagram
conventions Intro-5
elements of Intro-8
System catalog tables
for triggers 2-3
systrigbody 1-8
systriggers 1-8
systrigbody table
contents of 2-5
datakey column in 1-8
index 2-5
systriggers table
content of 2-4
indexes 2-4
querying 1-8
trigid column 1-9

T

TRACE command
output from 1-18
using 1-17
Trigger
and other SQL statements 2-36
benefits of 1-3
consists of 1-3
definition of 1-3
header information 1-8
in client/server environment 2-32
looking up 1-8
number on a table 2-8
preventing overriding 2-31
trigger name 1-4
uses for 1-4

Trigger event
definition of 1-4, 2-8
in CREATE TRIGGER statement 2-8
INSERT 2-17
privileges on 2-9
with cursor statement 2-9
Trigger name, syntax 2-11
Triggered action
action on triggering table 2-26
anyone can use 2-28
avoiding dependence on row order 2-15
cascading 2-16
clause 2-21
component of trigger 1-3
correlation name in 2-24, 2-27
definition of 1-3
FOR EACH ROW 2-15
in client/server environment 2-32
merged 2-15
occurs 1-5
preventing overriding 2-31
WHEN condition 2-21
Triggered action clause
action statements 2-22
syntax 2-21
WHEN condition 2-21
Triggered action list
AFTER 2-15
BEFORE 2-14
FOR EACH ROW 2-14
for multiple triggers 2-15
sequence of 2-14
Triggering statement
affecting multiple rows 2-15
component of trigger 1-3
execution of 2-10
guaranteeing same result 2-8
result of 2-22
types of 1-3
UPDATE 2-13
Triggering table
action on 2-26
and cascading triggers 2-30
trigid column, in systriggers table 1-9
Typographical conventions Intro-5

U

UPDATE clause, syntax 2-12
UPDATE REFERENCING clause
and FOR EACH ROW section 2-20
correlation name 2-19
syntax 2-19
UPDATE statement
as triggered action 2-22
as triggering statement 2-9, 2-12, 2-13
in trigger event 2-8
Update trigger, defining multiple 2-12
Using correlation names 2-24
Using triggers
automating changes 1-10
constructing before and after images 1-12
deriving data 1-14
implementing business rules 1-12
some applications 1-10

W

WHEN condition
in triggered action 2-21
restrictions 2-22
use of 2-21
WHERE CURRENT OF clause, impact on trigger 2-9

Symbols

--, adding comments with 1-6
{}, adding comments with 1-6