2001

Application of active rules to support database integrity constraints and view management

Suwanna Visavapattamawon

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Databases and Information Systems Commons

Recommended Citation

https://scholarworks.lib.csusb.edu/etd-project/1981

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
APPLICATION OF ACTIVE RULES TO SUPPORT
DATABASE INTEGRITY CONSTRAINTS AND VIEW MANAGEMENT

A Project
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Suwanna Visavapattamawon
June 2001
APPLICATION OF ACTIVE RULES TO SUPPORT DATABASE INTEGRITY CONSTRAINTS AND VIEW MANAGEMENT

A Project
Presented to the
Faculty of
California State University,
San Bernardino

by
Suwanna Visavapattamawon
June 2001

Approved by:

Dr. Josephine Mendoza, Computer Science
Dr. Arturo I. Concepcion
Dr. George M. Georgiou

4-18-01
Date
ABSTRACT

A conventional database system is a facility for storing large volumes of data and allowing multiple users to manipulate the data in an efficient and controlled approach. It supports limited forms of integrity constraints—primary key, unique key, foreign key, not null constraint and check constraint. However, in today's businesses, many applications require constraints well beyond these simple forms, such as the specification that all values must be within a certain range, or that a cumulative function on the data must produce a certain value.

Like a conventional database system, an "active" database can support all limited integrity constraints via an efficient mechanism—rule processing (or "trigger"). Additionally, an active database system provides other functionalities such as view management, data authorization, database auditing, event logging and many other database features. These functions are currently required in many commercial database management systems.

This project demonstrates the enforcement of integrity constraints in both the conventional and active database systems. The project implements a more complex user-defined
constraint, a complicated view and more detailed database auditing on the active database system. The performance of the declarative method will be compared with the trigger method using the following criteria—capabilities, limitations, flexibilities, data verification of an existing data, and execution time. The systems used for implementation are ORACLE 8 and SQL SERVER 7.0 database systems.
ACKNOWLEDGMENTS

I would like to thank my project advisor Professor Josephine Mendoza, Ph.D., who strongly supported me in developing my master project. I also thank my committee: Professor Arturo I. Concepcion, Ph.D. and Professor George M. Georgiou, Ph.D., for their valuable guidance and suggestions.

I would like to express my gratitude to Mr. Sumuscha Teesri, who was always my supporter during the several months that I worked on this project, Mr. Suchart Teesri, and my older sister in the USA who were also supportive in this project.

Finally, a special note of appreciation to my family members in Thailand--especially my father and mother whose encouragement and patience made my education possible and successful, and my sisters and brothers for taking care of my parents while I study in the USA.

Suwanna Visavapattamawon
To Mom, Dad and Ball
# TABLE OF CONTENTS

**ABSTRACT** ................................................................. iii

**ACKNOWLEDGMENTS** .................................................. v

**LIST OF TABLES** ...................................................... x

**LIST OF FIGURES** ................................................... xi

**CHAPTER ONE: SOFTWARE REQUIREMENT SPECIFICATION**

1.1 Introduction ......................................................... 1

1.2 Product Overview ................................................... 3

1.3 Product Perspective ................................................ 5

1.3.1 System Interfaces ................................................. 5

1.3.2 User Interfaces .................................................. 6

1.3.3 Hardware Interfaces ............................................. 6

1.3.4 Software Interfaces ............................................. 7

**CHAPTER TWO: DETAILED DESIGN**

2.1 The COMPANY Database ............................................. 8

2.2 Data Dictionary .................................................... 10

2.3 Integrity Constraints ............................................... 13

2.4 User-defined Constraints .......................................... 16

2.5 Repair Strategies .................................................. 17

2.5.1 Repair Strategies Using Triggers
               in the COMPANY1 Database .................................. 18

2.5.2 Repair Strategies Using the
               Declarative Method in the COMPANY2
               Database ....................................................... 19

**CHAPTER THREE: IMPLEMENTATION**

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Declarative Method in ORACLE 8 and SQL SERVER 7.0</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>The Trigger Method in ORACLE 8 and SQL SERVER 7.0</td>
<td>23</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Trigger Level</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1.1</td>
<td>Statement-level Trigger</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1.2</td>
<td>Row-level Trigger</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Trigger Timing</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2.1</td>
<td>BEFORE</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2.2</td>
<td>AFTER</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Triggers in ORACLE 8</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Triggers in SQL SERVER 7.0</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Triggers Defined for the COMPANY Database</td>
<td>40</td>
</tr>
<tr>
<td>3.6</td>
<td>Implementing COMPANY Triggers Using ORACLE 8</td>
<td>57</td>
</tr>
<tr>
<td>3.7</td>
<td>Implementing COMPANY Triggers Using SQL SERVER 7.0</td>
<td>67</td>
</tr>
<tr>
<td>3.8</td>
<td>Triggers in Auditing Databases</td>
<td>70</td>
</tr>
</tbody>
</table>

CHAPTER FOUR: VIEW MANAGEMENT

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>View Modification</td>
<td>76</td>
</tr>
<tr>
<td>4.2</td>
<td>The PROJECT_INFO View</td>
<td>79</td>
</tr>
<tr>
<td>4.3</td>
<td>A PROJECT_INFO View Modification in ORACLE 8</td>
<td>80</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementing the PROJECT_INFO View in ORACLE 8</td>
<td>85</td>
</tr>
<tr>
<td>4.5</td>
<td>A PROJECT_INFO View Modification in SQL SERVER 7.0</td>
<td>89</td>
</tr>
</tbody>
</table>
CHAPTER FIVE: ANALYSIS

5.1 Comparison Between Declarative and Trigger Methods

5.1.1 Implementing Integrity Constraints

5.1.2 Verifying the Existing Data

5.1.3 Ease of Use

5.1.4 Execution Time

5.1.5 Flexibility

5.1.6 Complex User-defined Constraints

5.1.7 View Maintenance

5.1.8 Database Auditing

5.2 Comparison Between Triggers in ORACLE 8 and SQL SERVER 7.0

5.2.1 Number of Triggers

5.2.2 Trigger Level

5.2.3 Trigger Timing

5.2.4 Trigger Combination

5.2.5 Referencing Data

5.2.6 Maintenance of Complicated Views

CHAPTER SIX: CONCLUSION AND FUTURE WORK

6.1 Conclusion

6.2 Future Work

APPENDIX A: TABLES AND VIEWS

The COMPANY1 Database in SQL SERVER 7.0

The COMPANY2 Database in SQL SERVER 7.0
LIST OF TABLES

Table 2.1 Data Dictionary in the COMPANY Database ..... 11
Table 2.2 Repair Strategies in the COMPANY Database ... 18
Table 3.1 Project Implementation ............................ 21
Table 3.2 The Trigger Mechanisms in ORACLE 8 and SQL SERVER 7.0 .................... 24
Table 3.3 Packages Used in Implementing Triggers ...... 57
Table 3.4 Implementing Triggers in SQL SERVER 7.0 ..... 68
Table 4.1 A PROJECT_INFO View Modification in SQL SERVER 7.0 and ORACLE 8 ............... 89
Table 5.1 Declarative Versus Trigger Methods .......... 91
Table 5.2 Statements Used to Enforce Constraints in the Declarative and Trigger Method --EMPLOYEE Table .......................... 93
Table 5.3 The Average Execution Time Using Declarative and Trigger Methods for Insert Operation .... 94
Table 5.4 The Execution Time Using Declarative and Trigger Methods for Delete Operation .... 95
Table 5.5 The Execution Time Using Declarative and Trigger Methods for Update Operation .... 96
Table 5.6 Triggers in ORACLE 8 Versus Triggers in SQL SERVER 7.0 ............................... 101
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Application Design from Conventional to Active Database System</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>System Interfaces</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>ER Diagram for the COMPANY Database</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Object Model for the COMPANY Database</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Referential Integrity Constraints in the COMPANY Database</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>A DELETE Operation on ORACLE 8</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>An Insert Operation on ORACLE 8</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>An Update Operation on ORACLE 8</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>A Delete Operation on SQL SERVER 7.0</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>An Insert Operation on SQL SERVER 7.0</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>An Update Operation on SQL SERVER 7.0</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>Delete on the EMPLOYEE Table</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>Insert on the EMPLOYEE Table</td>
<td>41</td>
</tr>
<tr>
<td>3.9</td>
<td>Update on the EMPLOYEE Table</td>
<td>43</td>
</tr>
<tr>
<td>3.10</td>
<td>Delete on the DEPARTMENT Table</td>
<td>46</td>
</tr>
<tr>
<td>3.11</td>
<td>Insert on the DEPARTMENT Table</td>
<td>47</td>
</tr>
<tr>
<td>3.12</td>
<td>Update on the DEPARTMENT Table</td>
<td>48</td>
</tr>
<tr>
<td>3.13</td>
<td>Insert on the LOCATION Table</td>
<td>49</td>
</tr>
<tr>
<td>3.14</td>
<td>Update on the LOCATION Table</td>
<td>50</td>
</tr>
<tr>
<td>3.15</td>
<td>Delete on the PROJECT Table</td>
<td>51</td>
</tr>
<tr>
<td>3.16</td>
<td>Insert on the PROJECT Table</td>
<td>52</td>
</tr>
<tr>
<td>3.17</td>
<td>Update on the PROJECT Table</td>
<td>53</td>
</tr>
</tbody>
</table>
Figure 4.7 An Insert of a New Project and Department Record into the PROJECT_INFO View ........... 83
Figure 4.8 An Update Operation on the PROJECT_INFO View .............................................. 85
Figure 4.9 Delete on the PROJECT_INFO View ............... 86
Figure 4.10 Insert on the PROJECT_INFO View ............. 87
Figure 4.11 Update on the PROJECT_INFO View ............. 88
Figure 5.1 Error Message Provided by System ............. 97
Figure 5.2 Error Message Provided by the Trigger Method ................................................. 98
Figure 5.3 The Combination of Triggers in ORACLE 8 ...104
CHAPTER ONE: SOFTWARE REQUIREMENT SPECIFICATION

1.1 Introduction

In a conventional database system, mechanisms for defining and enforcing integrity constraints are limited to specific types of constraints, such as uniqueness of keys and referential integrity constraints. Integrity constraints in a conventional database system are maintained either by rolling back any transaction that produces an inconsistent state or by disallowing or modifying operations that may produce an inconsistent state.

An improved approach to enforce constraints, especially user-defined constraints, is to have an active database system. The active database system uses an active rule to detect constraint violations and initiates database operations to restore consistency. It also allows developers to generate the flexible repair strategies to correct constraint violations.

Figure 1.1 shows the application design of conventional and active database systems. In a conventional database system, application programs perform data management, access, and modification functions by interacting with the database management systems (DBMS) through a program interface. In active database systems, an
active rule [1] [2] [5] [9] will be added between the schema and the applications in order to monitor the data, perform some actions automatically, audit the database, maintain views, enforce data management policies and so on. An active rule will make the size of applications smaller than those of a conventional database because some tasks can be done in the active rule parts instead of application parts.

The project implements the COMPANY database using two databases systems--SQL SERVER 7.0 [10] [11] [12] and ORACLE 8 [3] [4] [6] [7] [8] to illustrate enforcement of standard integrity constraints, user-defined constraints, view management and database auditing.
1.2 Product Overview

This project deals with two constraint types: (1) standard integrity constraints and (2) user-defined constraints.
The standard integrity constraints include:

- **PRIMARY KEY**: Column cannot be NULL and must have unique value. Each table can have only one primary key.

- **NOT NULL**: Column cannot be NULL.

- **CHECK**: A check clause specifies a condition restricting the value of the column.

- **UNIQUE**: Column must contain a unique value or is NULL. No two tuples may have the same value for the specified column.

- **FOREIGN KEY**: A constraint that maintains consistency among tuples of two relations (tables). For example, a tuple (record) in one table (referencing table) must refer to an existing tuple (record) in another table (referred table) [1].

A user-defined constraint is a (business) rule that is specified by the user.

The project demonstrates how these constraints are enforced in two different ways: (1) declarative method using the constraint clauses in the SQL Data Definition Language (DDL) and (2) triggers.

To compare the trigger and declarative methods, two versions of COMPANY database, COMPANY1 and COMPANY2 databases, will be created in SQL SERVER 7.0 and ORACLE 8.
The COMPANY1 database demonstrates the use of triggers and the COMPANY2 database illustrates the declarative methods.

Finally, a comparative analysis between the use of triggers and declarative methods is based on the ability and performance to enforce an integrity constraint when a violation occurs on primary key, not null, unique key, foreign key and check constraint. Additionally, the performance of triggers to enforce on user-defined constraints, view maintenance, and audit database will be considered. The analysis will also discuss the advantages, disadvantages and the limitations of using trigger and declarative methods.

1.3 Product Perspective

1.3.1 System Interfaces

Figure 1.2 shows the component of the system interfaces used in the project. The system interfaces are composed of a client computer connecting to the COMPANY database in both SQL SERVER 7.0 and ORACLE 8 on the server.
1.3.2 User Interfaces

The user interface is a SQL DML (DELETE/ INSERT/ UPDATE) in SQL SERVER 7.0 and ORACLE 8.

1.3.3 Hardware Interfaces

The hardware required to implement the databases are:

Server:

- AMD K6-2/450 3D NOW
- Hard drive 18 GB
- Physical memory 256 MB
Client:
- AMD K6-2/333
- Hard drive 2 GB
- Physical memory 128 MB

1.3.4 Software Interfaces

The software applications required to implement the database are:
- Windows 2000 Professional for client
- SQL SERVER 7.0
- ORACLE 8.0.5
CHAPTER TWO: DETAILED DESIGN

In this section, the Entity-Relationship ER diagram and an object model for the COMPANY database will be given. The data dictionary details the data information and includes all integrity constraints. SQL statements used to create the tables of the COMPANY database on both SQL SERVER 7.0 and ORACLE 8 can be found in Appendix A.

2.1 The COMPANY Database

The COMPANY database will keep information about employees working in a company. Each department must have at least one employee and only one manager, and can be located in many places. Each employee, a supervisee, has at most one direct supervisor, while a supervisor may supervise several employees. An employee need not work for a department and may work on several projects. Each project is controlled by only one department and a department can manage many projects.

There are two versions of the COMPANY databases--COMPANY1 and COMPANY2. The COMPANY1 will implement constraints by triggers. The COMPANY2 will use the declarative method. The entity relationship (ER) and the object model diagrams for the COMPANY database are shown in Figures 2.1 and Figure 2.2.
Figure 2.1 ER Diagram for the COMPANY Database
2.2 Data Dictionary

Table 2.1 shows the data dictionary of the COMPANY database. It includes the information on table names, column names, data types, constraints, and descriptions of column names.
<table>
<thead>
<tr>
<th>Table Name</th>
<th>Column</th>
<th>Data Type</th>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE</td>
<td>empno</td>
<td>varchar2(11)</td>
<td>primary key</td>
<td>Employee number uniquely identifies each employee.</td>
</tr>
<tr>
<td></td>
<td>fname</td>
<td>varchar2(20)</td>
<td>not null</td>
<td>Employee's first name.</td>
</tr>
<tr>
<td></td>
<td>mint</td>
<td>char(1)</td>
<td></td>
<td>Employee’s middle initial</td>
</tr>
<tr>
<td></td>
<td>lname</td>
<td>varchar2(25)</td>
<td>not null</td>
<td>Employee’s last name.</td>
</tr>
<tr>
<td></td>
<td>bdate</td>
<td>date</td>
<td>not null</td>
<td>Employee’s birthdate.</td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>varchar2(40)</td>
<td>not null</td>
<td>Employee’s home address.</td>
</tr>
<tr>
<td></td>
<td>tel</td>
<td>varchar2(20)</td>
<td></td>
<td>Employee’s home telephone number.</td>
</tr>
<tr>
<td></td>
<td>sex</td>
<td>char(1)</td>
<td>check -F: female-M: male.</td>
<td>The gender can be either male or female.</td>
</tr>
<tr>
<td></td>
<td>salary</td>
<td>number</td>
<td>check &gt; 0,</td>
<td>Employee’s yearly salary</td>
</tr>
<tr>
<td></td>
<td>ssn</td>
<td>varchar2(11)</td>
<td>foreign key</td>
<td>Employee number of the employee’s supervisor</td>
</tr>
<tr>
<td></td>
<td>dno</td>
<td>number</td>
<td>foreign key</td>
<td>The department number of the employee’s department.</td>
</tr>
<tr>
<td>Table Name</td>
<td>Column</td>
<td>Data Type</td>
<td>Constraint</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>------------</td>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>dno</td>
<td>number</td>
<td>primary key</td>
<td>Department number</td>
</tr>
<tr>
<td></td>
<td>dname</td>
<td>varchar2</td>
<td>not null, unique key</td>
<td>Department name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(40)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mgrssn</td>
<td>varchar2</td>
<td>foreign key</td>
<td>Employee number of the department's manager.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11)</td>
<td>(1-1:manages relationship)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCATION</td>
<td>dno</td>
<td>number</td>
<td>primary key</td>
<td>Department number</td>
</tr>
<tr>
<td>(multi-valued attributed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dlocation</td>
<td>varchar2</td>
<td></td>
<td>Department's location.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(30)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORKS_ON</td>
<td>empno</td>
<td>varchar2</td>
<td>primary key</td>
<td>Employee number of the employee who works on the project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11)</td>
<td>(M-N: relationship)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pnumber</td>
<td>number</td>
<td>Primary key</td>
<td>The project number that the employee works on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hours</td>
<td>number</td>
<td>check &gt; 0</td>
<td>The hours worked by the employee on the project.</td>
</tr>
<tr>
<td>PROJECT</td>
<td>pnumber</td>
<td>Number</td>
<td>primary key</td>
<td>Project number.</td>
</tr>
<tr>
<td></td>
<td>pname</td>
<td>varchar2</td>
<td>not null, unique key</td>
<td>Project name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(30)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dno</td>
<td>number</td>
<td>foreign key</td>
<td>Department number of the department that controls the project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(1-N: controls relationship)</td>
<td></td>
</tr>
</tbody>
</table>
2.3 Integrity Constraints

The integrity constraints that are implemented in the project are primary key, unique key, foreign key, not null and check constraint. The referential integrity constraints in the COMPANY database are illustrated in Figure 2.3 and explained in detail below. Primary keys are underlined.

Figure 2.3 Referential Integrity Constraints in the COMPANY Database
EMPILOYEE Table:

1. Primary key: Empno must be unique and cannot be null.
2. Check constraint: Sex is male, female or null (unknown).
3. Foreign key constraints:
   i. Attribute ssn in the EMPLOYEE table refers to a supervisor of the employee. The ssn can be null if the employee has no supervisor. Otherwise, the ssn value must match an empno value in the EMPLOYEE table.
   ii. Attribute dno refers to the department in which an employee works. The value of dno in the EMPLOYEE table must match a dno value in the DEPARTMENT table. If the employee does not work for any department, then the dno value must be null.

DEPARTMENT Table:

1. Primary key: Dno must be unique and cannot be null.
2. Foreign key constraint: Attribute mgrssn refers to a department manager who is an employee in the EMPLOYEE table. The value of mgrssn must match an empno value in the EMPLOYEE table. Since each department must have a manager, the mgrssn attribute cannot be null.
WORKS_ON Table:

1. Primary key (Composite key): Empno and pnumber must be unique. Both attributes cannot be null.

2. Foreign key constraints:
   i. Attribute empno refers to an employee who works on any project. The value of empno in the WORKS_ON table cannot be null and must match an empno value in the EMPLOYEE table.
   ii. Attribute pnumber refers to the project number on which an employee is working. The value of pnumber cannot be null and must match a pnumber value in the PROJECT table.

PROJECT Table:

1. Primary key: Pnumber must be unique and cannot be null.

2. Foreign key constraint: Attribute dno refers to the department that controls the project. The value of a dno must match a dno value in the DEPARTMENT table and cannot be null since each project must be controlled by a department.

LOCATION Table:

1. Primary key (Composite key): Dno and dlocation must be unique and both attributes cannot be null.
2. Foreign key constraint: Attribute \texttt{dno} refers to a department number of a department. The value of \texttt{dno} must match a dno value in the DEPARTMENT table.

2.4 User-defined Constraints

User-defined constraints for the COMPANY database are:
- The salary of an employee, who is not a manager, cannot be increased by more than 15%.
- Each supervisee cannot have a salary more than his/her supervisor's salary.
- Each employee, who is not a manager, cannot work in more than 10 projects.
- Employees must be at least eighteen years of age and not more than sixty-five years old.
- Each employee, who is more than sixty years old, will not get a raise in salary.
- Each employee cannot have a salary more than the manager.

These user-defined constraints cannot be enforced by using CHECK constraints available in the declarative method. However, these constraints can be enforced using triggers.
2.5 Repair Strategies

Integrity constraint violations in the database system can happen during insert, update or delete operations. When a violation occurs, a repair strategy must be used to correct the violation. The different repair strategy options are:

- ALERT (message): signals the user or application that a constraint has been violated.
- RESTRICT: undoes the operation that caused the constraint's violation.
- ROLLBACK: causes the current transaction to be rolled back.
- CASCADE: applies to UPDATE or DELETE events. Whenever an update or delete occurs on the referred table, all referencing tuples are also updated or deleted appropriately.
- Set to null: sets the value of the referencing attribute that refers to the referred table to null.
- Set to default: sets the referencing attribute referring to the referred table to a default value.

Table 2.2 shows the summary of the repair strategies used to implement in this project.
Table 2.2 Repair Strategies in the COMPANY Database

<table>
<thead>
<tr>
<th>Constraint Types</th>
<th>Repair Strategies</th>
<th>COMPANY1 (Triggers)</th>
<th>COMPANY2 (Declarative Methods)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key</td>
<td>ALERT and ROLLBACK</td>
<td>ALERT</td>
<td></td>
</tr>
<tr>
<td>Unique Key</td>
<td>ALERT and ROLLBACK</td>
<td>ALERT</td>
<td></td>
</tr>
<tr>
<td>Not null</td>
<td>ALERT and ROLLBACK</td>
<td>ALERT</td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td>ALERT and ROLLBACK</td>
<td>ALERT</td>
<td></td>
</tr>
<tr>
<td>REFERENTIAL INTEGRITY</td>
<td>UPDATE CASCADE</td>
<td>UPDATE RESTRICT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE CASCADE</td>
<td>DELETE RESTRICT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET TO NULL</td>
<td>DELETE CASCADE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET TO DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User-defined constraint</td>
<td>ALERT and ROLLBACK</td>
<td>ALERT and ROLLBACK</td>
<td></td>
</tr>
</tbody>
</table>

2.5.1 Repair Strategies Using Triggers in the COMPANY1 Database

In the COMPANY1 database, primary key constraint, check constraint and foreign key constraint will be checked on INSERT, UPDATE, or DELETE. The repair strategies used are (1) alert (2) rollback (3) cascade (4) set to null and (5) set to a default value.

Primary key constraint:

When a primary key is inserted or updated, it verifies that the value of an inserted or updated primary key is unique and not null. If the new primary key value is duplicated or is null, an alert message will be displayed,
and the transaction that causes a violation will be rolled back.

Check Constraint:
For a check constraint violation, an alert message will be displayed and the transaction will be rolled back.

Foreign key constraint:
Whenever a foreign key value is inserted or updated, it will be verified that the inserted or updated value exists as a primary key value in the referred base table or it is null. An alert message will be displayed and the transaction will be rolled back on a violation.

The set to null and set to a default strategies will be used to retain referential integrity constraint on a delete operation. Whenever referenced data is deleted, the associated dependent data is set to null or a default value.

Finally, the cascade strategy is used to maintain referential integrity on an update operation. An update on the referenced table will also update corresponding tuples in the referencing table.

2.5.2 Repair Strategies Using the Declarative Method in the COMPANY2 Database

Three types of repair strategies can be used in SQL SERVER 7.0 and ORACLE 8:
- **ALERT:** When a violation occurs on insert, update or delete, the system will alert and display error message.

- **RESTRICT:** The RESTRICT option will not allow an update on the referenced table if the updated value is referred in the referencing table.

- **CASCADE:** The CASCADE option is only for a delete cascade in the ORACLE 8. If the CASCADE option is declared, all related values in the referencing table that refers to the deleted primary key value in the referred table will also be deleted.
CHAPTER THREE: IMPLEMENTATION

Two versions of the COMPANY databases are created and implemented using two different database management systems. SQL SERVER 7.0 is used to represent the relational version and ORACLE 8 to illustrate the object-relational version (See Table 3.1).

Table 3.1 Project Implementation

<table>
<thead>
<tr>
<th>Version</th>
<th>COMPANY1 (Trigger Method)</th>
<th>COMPANY2 (Declarative Method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>SQL SERVER 7.0</td>
<td>SQL SERVER 7.0</td>
</tr>
<tr>
<td>Object</td>
<td>ORACLE 8</td>
<td>ORACLE 8</td>
</tr>
</tbody>
</table>

3.1 The Declarative Method in ORACLE 8 and SQL SERVER 7.0

The processes to employ the declarative method in ORACLE 8 and SQL SERVER 7.0 are very similar. First, a table needs to be created. Second, any required constraint may be declared during or after table definition. Finally, the system will enforce the predefined constraints.

The CREATE TABLE statement in the EMPLOYEE table illustrates the integrity constraints using the declarative method. These constraints include the primary key, unique key, foreign key, not null, and check constraints as shown in the following section:
### EMPLOYEE Table:

```sql
CREATE TABLE "COMPANY2".employee
(empno VARCHAR2(11) NOT NULL,
fname VARCHAR2(20) NOT NULL,
mint CHAR(1),
lname VARCHAR2(25) NOT NULL,
bdate DATE NOT NULL,
address VARCHAR2(40) NOT NULL,
tel VARCHAR2(20),
sex CHAR(1),
salary NUMBER(8) NOT NULL,
ssn VARCHAR2(11),
dno NUMBER(4));
```

/*Adding Constraints to the EMPLOYEE table. */

```sql
ALTER TABLE employee
ADD CONSTRAINT emp_empno_pk PRIMARY KEY (empno);

ALTER TABLE employee
ADD CONSTRAINT emp_sex_ck
CHECK (sex in ('F', 'M', 'f', 'm'));

ALTER TABLE employee
ADD CONSTRAINT emp_salary_ck
CHECK (salary >= 0);

ALTER TABLE employee
ADD CONSTRAINT emp_ssn_fk FOREIGN KEY (ssn)
```
REFERENCES employee (empno) ON DELETE CASCADE;

ALTER TABLE employee
ADD CONSTRAINT emp_dno_fk FOREIGN KEY (dno)
REFERENCES department (dno) ON DELETE CASCADE;

The difference between SQL SERVER 7.0 and ORACLE 8 declarative method is that the SQL SERVER 7.0 does not allow DELETE CASCADE, but ORACLE 8 does.

3.2 The Trigger Method in ORACLE 8 and SQL SERVER 7.0

A trigger is a special stored procedure that neither accepts arguments nor returns code. It will be fired automatically when a delete, insert or update operation occurs on a table. Before executing triggers, the system will check all predefined constraints. If the predefined constraint operations fail, triggers will not be executed. The trigger normally will be used to implement constraints that cannot be applied by the system, such as the cross-reference user-defined constraints.

Unlike the declarative method, the trigger method will not declare the integrity constraints during table creation. These constraints are declared in the CREATE TRIGGER statement after generating a table. The following CREATE TABLE statement shows that no constraint is declared during table definition.
DEPARTMENT Table:

CREATE TABLE "COMPANY1".department
(dno int,
 dname VARCHAR2(40),
 mgrssn VARCHAR2(11));

ORACLE 8 and SQL SERVER 7.0 use different mechanisms to implement triggers. The following sections describe how to implement triggers in ORACLE 8 and SQL SERVER 7.0. Table 3.2 shows the trigger mechanism in both systems.

Table 3.2 The Trigger Mechanisms in ORACLE 8 and SQL SERVER 7.0

<table>
<thead>
<tr>
<th>Mechanisms</th>
<th>ORACLE 8</th>
<th>SQL SERVER 7.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger level</td>
<td>- statement-level</td>
<td>- statement-level</td>
</tr>
<tr>
<td></td>
<td>- row-level</td>
<td></td>
</tr>
<tr>
<td>Trigger Timing</td>
<td>BEFORE and AFTER</td>
<td>AFTER</td>
</tr>
<tr>
<td>Record/Table References</td>
<td>:new and :old</td>
<td>INSERTED and</td>
</tr>
<tr>
<td></td>
<td>pseudorecord</td>
<td>DELETED Table</td>
</tr>
</tbody>
</table>

3.2.1 Trigger Level

A trigger can specify the number of times the trigger action to be executed: (1) statement-level trigger: once for each SQL DML statement and (2) row-level trigger: once for each row.
3.2.1.1 Statement-level Trigger

The statement-level trigger fires once for each SQL DML statement regardless of the number of rows that the SQL DML statement affects. For example, if an UPDATE statement affects 10 rows in the table, the statement-level trigger fires on the table only one time. The statement-level trigger, therefore, is not often used for data-related activities. They are normally used to enforce additional security that applies for a table. For example, the statement-level trigger can be used to verify an authorized user who modifies the EMPLOYEE table.

3.2.1.2 Row-level Trigger

Unlike the statement-level trigger, the row-level trigger fires once for each row affected by the SQL DML statements. If an UPDATE statement affects 10 rows, then the trigger will fire 10 times. The row-level trigger is normally used for data-related activities and auditing database. For example, the company policy does not allow a salary increase of more than 15%. When all employees get a $1000 raise, the row-level trigger will check whether a $1000 increase is more than 15% raise in every EMPLOYEE record.
3.2.2 Trigger Timing

A trigger can specify whether it can be executed before or after the SQL DML statement. Timing can be defined by using "BEFORE" or "AFTER" options in the CREATE TRIGGER command. Both BEFORE and AFTER are applicable in the statement-level trigger and the row-level trigger.

3.2.2.1 BEFORE

In the BEFORE trigger, the trigger action will be performed before executing the SQL DML statement. The BEFORE trigger is used to decide whether the SQL DML statement should be executed. It helps to eliminate unnecessary processing of the SQL DML statement if the SQL DML causes a violation. For example, the BEFORE trigger can be used to verify the uniqueness of a primary key before allowing an insert of a new record. The new record will be inserted when its primary key is unique, otherwise, the transaction will be rolled back.

3.2.2.2 AFTER

The AFTER trigger timing performs the trigger action after the SQL DML statement is executed. It is used when a task needs to be done after the SQL DML is completed. For instance, in a distributed database system, the AFTER trigger can be used to coordinate a synchronous distributed database transaction. For example, when a modification is
done to the central database, the trigger can be used to verify that same modification is done on the replicated database. Additionally, when auditing a database, the AFTER trigger will be fired to keep the change in an audit table.

3.3 Triggers in ORACLE 8

ORACLE 8 provides both statement and row level triggers as well as BEFORE and AFTER trigger timings. The combination of ORACLE 8 triggers can be classified as follows.

1. BEFORE statement-level trigger
2. BEFORE row-level trigger
3. AFTER statement-level trigger
4. AFTER row-level trigger

Many triggers can perform a single SQL DML operation. For example, an UPDATE operation can be implemented using a BEFORE statement-level trigger, a BEFORE row-level trigger, and an AFTER statement-level trigger at the same time. When several triggers fire on a SQL DML statement, the firing order of multiple triggers is as follows:

1. SQL DML statement is issued.
2. Any BEFORE statement-level trigger is executed.
3. For each row affected by the triggering SQL statement.
   a. Any BEFORE row-level trigger is executed.
b. A row is locked and changed. An integrity constraint is checked. The lock is not released until the transaction is committed.

c. All AFTER row-level triggers are executed.

4. Any AFTER statement-level triggers are executed.

The following is the syntax of creating a trigger:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} [INSTEAD OF] triggering_event
ON table_name
[FOR EACH ROW [WHEN triggering restriction]]
trigger body
```

Syntax Description:

- **trigger_name**: The name of a trigger.
- **BEFORE | AFTER**: The time when a trigger is to be executed before or after the triggering statement.
- **triggering event**: The INSERT, UPDATE, or DELETE SQL statement that causes the trigger to be fired.
- **INSTEAD OF**: The trigger option for the view objects.
- **FOR EACH ROW**: The option determines whether the trigger is a row or
statement trigger. For the FOR EACH ROW option, a trigger fires once in each row that is affected by the triggering statement. Without this specification, the trigger fires once regardless of the number of affected rows. This trigger is called a statement trigger.

Triggering restriction The restriction specifies a boolean logical expression that must be true for the trigger to fire.

trigger body The PL/SQL or SQL statements to define the tasks or actions to execute when the trigger fires.

When a trigger is created, ORACLE 8 uses two pseudorecords--:old and :new-- to reference a field in the affected rows and to test for a violation. These pseudorecords can be defined only in the row-level trigger. They are generated automatically during any SQL DML operation such as DELETE, INSERT, or UPDATE. A specific
field can be referred to by using :old.column_name or :new.column_name.

In the delete operation, the :old refers to a deleted record. The :old.column_name refers to the deleted value in the specific field such as :old.dno and :old.dname. An example of deleting a department is shown in Figure 3.1. After deleting a department (dno=4), the :old will keep the information of deleted department, which is the department number 4.
Figure 3.1 A DELETE Operation on ORACLE 8

Before DELETE operation

DEPARTMENT:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>4</td>
<td>Environment</td>
<td>777888999</td>
</tr>
</tbody>
</table>

:old

DEPARTMENT:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
</tbody>
</table>

Command executes:

```
DELETE FROM DEPARTMENT WHERE dno = 4;
```

After DELETE operation

DEPARTMENT:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>4</td>
<td>Environment</td>
<td>777888999</td>
</tr>
</tbody>
</table>

:old

DEPARTMENT:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Environment</td>
<td>777888999</td>
</tr>
</tbody>
</table>

In the insert operation, the :new refers to an inserted record. The :new.column_name refers to the new inserted value of a specific field such as :new.dno, :new.dname and :new.mgrssn. Figure 3.2 demonstrates the processes of triggers on an INSERT operation in the
DEPARTMENT table. After inserting a new department \((dno=5)\), the :new will keep the information of this department record.

**Figure 3.2 An Insert Operation on ORACLE 8**

Before INSERT operation

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
</tbody>
</table>

: new

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Command executes:

```
INSERT INTO DEPARTMENT
VALUES (5, 'Marketing', '123456789');
```

After INSERT operation

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>5</td>
<td>Marketing</td>
<td>123456789</td>
</tr>
</tbody>
</table>

: new

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Marketing</td>
<td>123456789</td>
</tr>
</tbody>
</table>

:new = (5, 'Marketing', '123456789')
:new.dno = 5
:new.dname = Marketing
:new.mgrssn = 123456789
For the update operation, the :old.column_name refers to the old value in the field before any change occurs. The :new.column_name refers to the new value after the field is modified. The processes of trigger on UPDATE operation demonstrate in Figure 3.3. After updating the name of department, the :old keeps the information of MARKETING department. At the same time, the :new will keep the information of ACCOUNTING department.
Figure 3.3 An Update Operation on ORACLE 8

Before UPDATE operation

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>5</td>
<td>Marketing</td>
<td>123456789</td>
</tr>
</tbody>
</table>

:old

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
</table>

:new

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
</table>

Command executes:

```
UPDATE DEPARTMENT
SET dname = 'Accounting'
WHERE dno = 5;
```

After UPDATE operation

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>5</td>
<td>Accounting</td>
<td>123456789</td>
</tr>
</tbody>
</table>

:old

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Marketing</td>
<td>123456789</td>
</tr>
</tbody>
</table>

:new

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
</table>

:old = (5, 'Marketing', '123456789')
:old.dno = 5
:old.dname = Marketing
:old.mgrssn = 123456789

:new = (5, 'Accounting', '123456789')
:new.dno = 5
:new.dname = Accounting
:new.mgrssn = 123456789
3.4 Triggers in SQL SERVER 7.0

In SQL SERVER 7.0, the row-level trigger is not available. This means every trigger will be executed only in the statement-level trigger. This trigger executes only one time against a given table after a SQL DML operation is performed. If there is more than one row affected by the SQL DML statement, only the last row will be validated. To overcome this problem, the trigger has to perform a DELETE, INSERT, or UPDATE in a loop.

SQL SERVER 7.0 provides only the AFTER trigger timing. This means it allows the SQL DML statement to be executed first, and then the trigger performs an execution. Triggers in SQL SERVER 7.0 also can be performed on more than one field in a table. For example, one trigger executes on an update on the empno field while another trigger fires the same operation on the dno field.

The syntax of creating a trigger in SQL SERVER 7.0 is:

```sql
CREATE TRIGGER [owner.]trigger_name
ON [owner.] table_name
FOR {INSERT, UPDATE, DELETE}
AS sql_statements
```

Syntax Description:

trigger_name The name of the trigger.
table_name The name of the table.

INSERT, UPDATE, The action that initiates trigger to
DELETE be executed.

sql_statements SQL statements to define the tasks
to execute when the trigger is
fired. The sql_statements will be
enclosed in BEGIN and END
delimiters.

When a trigger is created, SQL SERVER 7.0 will
generate special tables--DELETED and INSERTED tables. These
tables are used to change the main table back to the
previous state if any violation occurs. After the trigger
completes, these two tables will no longer exist.

For a delete operation, SQL SERVER 7.0 will remove the
deleted row from the main table to the DELETED table as
seen in Figure 3.4. The department number (dno =4 ) will be
kept in the DELETED table.
Figure 3.4 A Delete Operation on SQL SERVER 7.0

Before DELETE operation

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>4</td>
<td>Environment</td>
<td>777888999</td>
</tr>
</tbody>
</table>

DELETED:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Command executes:

DETECT FROM DEPARTMENT WHERE dno = 4;

After DELETE operation

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>333444555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
</tbody>
</table>

DELETED:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Environment</td>
<td>777888999</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DELETED.dno = 4
DELETED.dname = Environment
DELETED.mgrssn = 777888999
For an insert operation, SQL SERVER 7.0 will copy a new row to an INSERTED table (see Figure 3.5). In case of an update operation, the old row will be copied into the DELETED table first and the new updated row will be copied into the INSERTED table as shown in Figure 3.6.

Figure 3.5 An Insert Operation on SQL SERVER 7.0

<table>
<thead>
<tr>
<th>Before INSERT operation</th>
<th>DEPARTMENT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNO</td>
<td>DNAME</td>
</tr>
<tr>
<td>1</td>
<td>Research</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
</tr>
</tbody>
</table>

INSERTED:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

Command executes:

```
INSERT INTO DEPARTMENT
VALUES (5, 'Marketing', '123456789');
```

After INSERT operation

<table>
<thead>
<tr>
<th>DEPARTMENT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

INSERTED:

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Marketing</td>
<td>123456789</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

| INSERTED.dno = 5 |
| INSERTED.dname = Marketing |
| INSERTED.mgrssn = 123456789 |
Figure 3.6 An Update Operation on SQL SERVER 7.0

<table>
<thead>
<tr>
<th>Before UPDATE operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT:</td>
</tr>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>DELETED:</td>
</tr>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>INSERTED:</td>
</tr>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Command executes:

```
UPDATE DEPARTMENT
SET dname = 'Accounting'
WHERE dno = 5;
```

<table>
<thead>
<tr>
<th>After UPDATE operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT:</td>
</tr>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>DELETED:</td>
</tr>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>INSERTED:</td>
</tr>
<tr>
<td>DNO</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

DELETED.dno = 5
DELETED.dname = Marketing
DELETED.mgrssn = 123456789

INSERTED.dno = 5
INSERTED.dname = Accounting
INSERTED.mgrssn = 123456789

39
3.5 Triggers Defined for the COMPANY Database

The following figures show the flowcharts of implementing triggers to enforce integrity constraints in the COMPANY database (Figure 3.7 - Figure 3.19).

Figure 3.7 Delete on the EMPLOYEE Table
Figure 3.8 Insert on the EMPLOYEE Table

**INSERT EMPLOYEE:**

1. **START**
2. Can insert this employee record, but need to check all integrity constraints
   - Does this employee exist in the table [empno == empno in other records]
     - yes
       - A primary key violation occurs. Display error message and rollback transaction.
     - no
6. Is this employee's empno null [empno is null]
   - yes
     - A not null violation occurs. Display error message and rollback transaction.
   - no
     - Is this employee's first name null [fname is null]
       - yes
         - A not null violation occurs. Display error message and rollback transaction.
       - no
         - Is this employee's last name null [lname is null]
           - yes
             - A not null violation occurs. Display error message and rollback transaction.
           - no
             - Is this employee's birthdate null [bdate is null]
               - yes
                 - A not null violation occurs. Display error message and rollback transaction.
               - no
                 - Is this employee's age not between 18 and 65 [(sysdate - bdate) < 18 or (sysdate - bdate) > 65]
                   - yes
                     - A user-defined constraint violation occurs. Display error message and rollback transaction.
                   - no
                     - Is this employee's address null [address is null]
                       - yes
                         - A not null violation occurs. Display error message and rollback transaction.
                       - no
                         1

```
INSERT EMPLOYEE with
empno = x, fname = 'Sam', mint = 'K',
lname = 'King', bdate = 12/28/1976,
address = '2345 Hollywood blvd., LA',
tel = '323-660-8999', sex = 'M', salary = 33000,
ssn = y, dno = 1
```

**END**
Figure 3.9 Update on the EMPLOYEE Table

UPDATE EMPLOYEE:

START

Update EMPLOYEE set dno = Y where empno = X

Can update this employee, but need to check integrity constraints

Is this employee exist in table

no

Update cannot be done

yes

Does this employee exist in table

no

Is this employee number updated

[update(empno)]

yes

Is it updated to null

[empno is null]

no

Is it updated to an existing employee

[empno == empno in other records]

yes

A not null violation occurs, Display error message and rollback transaction.

no

Is this employee a supervisor

[empno == ssn in other records]

yes

Update those supervisors (ssn) to the new employee number

no

Is this employee a manager of department

[empno == mgrssn in DEPARTMENT]

yes

Update the manager (mgrssn) to the new employee number

no

Does this employee work in project

[empno == empno in WORKS_ON]

yes

Update employee number in work records of this employee to the new employees number

no

Is this employee's first name updated

[update(fname)]

yes

Is it updated to null

[frame is null]

no

A not null violation occurs, Display error message and rollback transaction.

no

END
A not null violation occurs. Is this employee's name updated? yes yes Is it updated to null? yes [name is null] A not null violation occurs. Display error message and rollback transaction. END

Is this employee's last name updated? yes [update(lname)] Is the age of employee not between 18 and 65? yes [sysdate - bdate] < 18 or (sysdate - bdate) > 65] A user-defined constraint violation occurs. Display error message and rollback transaction. END

Is this employee's birthdate updated? yes [update(bdate)] A not null violation occurs. Display error message and rollback transaction. END

Is this employee's address updated? yes [update(address)] A not null violation occurs. Display error message and rollback transaction. END

Is this employee's gender updated? yes [update(sex)] (If the sex is not null) A check constraint violation occurs. Display error message and rollback transaction. END

Is this employee's supervisor updated? yes [update(ssn)] (If the employee's supervisor is not null) A foreign key violation occurs. Display error message and rollback transaction. END

Is this department updated? yes [update(dno)] (If the employee's department is not null) A foreign key violation occurs. Display error message and rollback transaction. END

44
A not null violation occurs. Display error message and rollback transaction.

A check constraint violation occurs. Display error message and rollback transaction.

A user-defined constraint violation occurs. Display error message and rollback transaction.

A user-defined constraint violation occurs. Display error message and rollback transaction.

A user-defined constraint violation occurs. Display error message and rollback transaction.

A user-defined constraint violation occurs. Display error message and rollback transaction.

A user-defined constraint violation occurs. Display error message and rollback transaction.

Keep the change of updating this employee's salary in the audit table.

Update this employee record.

END.
DELETE DEPARTMENT:

START

Does this department exist in the table?

no

Deletion cannot be done.

yes

Can delete this department, but need to check for integrity constraints.

Is this department a default department (10) [dno == 10]?

yes

Deletion cannot be done. An alert message will be displayed and transaction is rollback.

no

Does this department control projects [dno == dno in other records in PROJECT]?

yes

Set the department number in project records to a default department (dept 10) and keep the change in audit table.

no

Does this department have employee working on it [dno == dno in other records in EMPLOYEE]?

yes

Set the department number in employee records to null and keep the change in audit table.

no

Delete all location records of this department

Delete this department record

END

END

Figure 3.10 Delete on the DEPARTMENT Table
Figure 3.11 Insert on the DEPARTMENT Table

**INSERT DEPARTMENT:**

```
START

Can insert this record, but need to check all integrity constraints

Does this department exist in the table
  yes
  A primary key violation occurs. Display error message and rollback transaction.
  no
  Is this department number null [dno is null]
    yes
    A not null violation occurs. Display error message and rollback transaction.
    no
    Is the name of department null [dname is null]
      yes
      A not null violation occurs. Display error message and rollback transaction.
      no
      Is the name of department an existing name [dname == dname in other records]
        yes
        A unique key violation occurs. Display error message and rollback transaction.
        no
        (If the manager of this department is not null)
          Is this department manager not an existing employee
            yes
            A foreign key violation occurs. Display error message and rollback transaction.
            no
            Insert this department record

  END
```

Insert DEPARTMENT with dno = B, dname = 'Research', mgrssn = X
Figure 3.12 Update on the DEPARTMENT Table

```
UPDATE DEPARTMENT:

START

no

Does this department exist in table?

yes

Can update this department record, but need to check integrity constraints.

no

Is the department number updated?

[update(dno)]

yes

Is it updated to null?

[dno is null]

yes

A not null violation occurs. Display error message and rollback transaction.

no

no

Is it updated to an existing department?

[dno == dno in other records]

yes

A primary key violation occurs. Display error message and rollback transaction.

no

no

Does this department have employee working on it?

[dno == dno in other records in EMPLOYEE]

yes

Update department number in employee records to the new department.

no

no

Does this department control any project?

[dno == dno in other records in PROJECT]

yes

Update department number in project records to the new department.

no

no

Update department number in location records to the new department.

Is the name of department updated?

[update(dname)]

yes

Is it updated to null?

[dno is null]

yes

A not null violation occurs. Display error message and rollback transaction.

no

no

Is it updated to an existing name?

[dname = dname in other records]

yes

A unique key violation occurs. Display error message and rollback transaction.

no

no

Is a manager of this department updated?

[update(mgrssn)]

yes

(IF a manager of this department is not null)

Is this manager not an existing employee?

[mgrssn != empno in EMPLOYEE]

yes

A foreign key violation occurs. Display error message and rollback transaction.

no

no

Update this department record.

END

END
```

48
Figure 3.13 Insert on the LOCATION Table

**INSERT LOCATION:**

1. **START**
   - Can Insert this record, but need to check all integrity constraints

2. Is this department not an existing department?
   - yes
     - [dno ! = dno in other records in DEPARTMENT]
   - no

3. Is this department number null?
   - yes
     - [dno is null]
   - no

4. Is this department location null?
   - yes
     - [dlocation is null]
   - no

5. Do both department and location exist in table?
   - yes
     - [dno and dlocation = dno and dlocation in other records]
   - no

6. Insert this department location record

7. **END**

Insert Location with dno = A and dlocation = 'San Bernardino'

- A foreign key violation occurs. Display error message and rollback transaction.

- A not null violation occurs. Display error message and rollback transaction.

- A primary key violation occurs. Display error message and rollback transaction.

**END**
Figure 3.14 Update on the LOCATION Table

UPDATE LOCATION:

START

Does this department location record exist in table?

no Is department number updated?

no Is it updated to null?

no A not null violation occurs. Display error message and rollback transaction.

yes A primary key violation occurs. Display error message and rollback transaction.

yes Is it updated to an existing department?

no [dno == dno in DEPARTMENT]

yes A foreign key violation occurs. Display error message and rollback transaction.

yes Do both department and location exist in table?

no [dno and dlocation == dno and dlocation in other records]

yes [dno and dlocation == dno and dlocation in other records]

no Is department location updated?

no [updated(dlocation)]

yes A not null violation occurs. Display error message and rollback transaction.

yes A primary key violation occurs. Display error message and rollback transaction.

yes Are department and location updated?

no [updated(dno) or updated(dlocation)]

yes Do both department and location exist in table?

no [dno and dlocation == dno and dlocation in other records]

yes [dno and dlocation == dno and dlocation in other records]

no Update cannot be done

END
Figure 3.15 Delete on the PROJECT Table

DELETE PROJECT:

START

Does this project exist in the table

no

Deletion cannot be done

yes

Delete PROJECT with pnumber = M

Can delete this project, but need to check for integrity constraints

Does this project have employees working on it [pnumber = pnumber in other records in WORKS_ON]

no

Delete work records of this project

yes

Delete this project

keep the information of deleted project in the audit table

END
**Figure 3.16 Insert on the PROJECT Table**

**INSERT PROJECT:**

1. **START**
   - Insert PROJECT with `pnumber = N`, `pname = Project_1`, `dno = A`

2. **Does this project exist in the table?**
   - [pnumber == pnumber in other records]
      - **Yes**
      - A primary key violation occurs. Display error message and rollback transaction.
      - **END**
   - **No**

3. **Is this project number null?**
   - [pnumber is null]
      - **Yes**
      - A not null violation occurs. Display error message and rollback transaction.
      - **END**
      - **No**

4. **Is the name of project null?**
   - [pname is null]
      - **Yes**
      - A not null violation occurs. Display error message and rollback transaction.
      - **END**
      - **No**

5. **Does this project's name exist in the table?**
   - [pname == pname in other records]
      - **Yes**
      - A unique key violation occurs. Display error message and rollback transaction.
      - **END**
      - **No**

6. **Is this department null?**
   - [dno is null]
      - **Yes**
      - A not null violation occurs. Display error message and rollback transaction.
      - **END**
      - **No**

7. **Is this department an existing department?**
   - [dno == dno in other records in DEPARTMENT]
      - **Yes**
      - A foreign key violation occurs. Display error message and rollback transaction.
      - **END**
      - **No**

8. **Insert this project record.**

9. **END**
Figure 3.17 Update on the PROJECT Table

UPDATE PROJECT:

START

Does this project exist in table?

yes  Can update this project record, but need to check all integrity constraints

no  Update cannot be done

END

Is project number updated?

{update(pnumber)}

yes  A not null violation occurs. Display error message and rollback transaction.

no  Is it updated to null?

{pnumber is null}

yes  Display error message and rollback transaction.

no  Is it updated to an existing project number?

{pnumber == pnumber in other records}

yes  A primary key violation occurs. Display error message and rollback transaction.

no  Does this project have employees working on it?

{pnumber == pnumber in other records in WORKS_ON}

yes  Update project in works_on records to the new project number

no  Update this project record

END

Is the project's name updated?

{update(pname)}

yes  A not null violation occurs. Display error message and rollback transaction.

no  Is it updated to null?

{pname is null}

yes  A unique key violation occurs. Display error message and rollback transaction.

no  Is it updated to an existing name?

{pname == pname in other records}

yes  Display error message and rollback transaction.

no  Is a department number updated?

{update(dno)}

yes  A foreign key violation occurs. Display error message and rollback transaction.

no  Update this project record

END

END

53
Figure 3.18 Insert on the WORKS_ON Table

INSERT WORKS_ON:

START

Can insert this record, but need to check all integrity constraints

Does this employee not exist in the table

Yes

A foreign key violation occurs. Display error message and rollback transaction.

No

Is this employee number null

Yes

A not null violation occurs. Display error message and rollback transaction.

No

Is this project number null

Yes

A not null violation occurs. Display error message and rollback transaction.

No

Does this project not exist in the table

Yes

A foreign key violation occurs. Display error message and rollback transaction.

No

Do both employee number and project number exist in the table

Yes

A primary key violation occurs. Display error message and rollback transaction.

No

Is this employee's work hour less than zero

Yes

A check constraint violation occurs. Display error message and rollback transaction.

No

(If the employee is not a manager)

Does this employee work in more than 10 projects

Yes

A user-defined constraint violation occurs. Display error message and rollback transaction.

No

Insert this work record

END
Figure 3.19 Update on the WORKS_ON Table

UPDATE WORKS_ON:

START

Update WORKS_ON set pnumber = M
where empno = X and pnumber = N

Can update this employee work record on record, but need to check integrity constraints

Is employee number updated [update(empno)]

Is it updated to null [pnumber is null]

A foreign key violation occurs. Display error message and rollback transaction.

Is it updated to an existing employee
[empno == empno in other records in EMPLOYEE]

A primary key violation occurs. Display error message and rollback transaction.

Is both employee number and project number exist in table
[empno and pnumber == empno and pnumber in other records]

A primary key violation occurs. Display error message and rollback transaction.

Is project number updated [update(pnumber)]

Is it updated to null [pnumber is null]

A not null violation occurs. Display error message and rollback transaction.

Is it not updated to an existing project
[pnumber == pnumber in other records in PROJECT]

A foreign key violation occurs. Display error message and rollback transaction.

Is both employee number and project number exist in table
[empno and pnumber == empno and pnumber in other records]

A primary key violation occurs. Display error message and rollback transaction.

Update cannot be done.
is employee number or project number updated [update(empno) or update(pnumber)]

Do both employee number and project number exist in table [empno and pnumber == empno and pnumber in other records]

A primary key violation occurs. Display error message and rollback transaction.

Is employee work hour updated [update(hour)]

Is employee work hour less than zero [hour < 0]

A check constraint violation occurs. Display error message and rollback transaction.

If this employee is not a manager does this employee work in more than 10 projects [total(project) > 10]

Update this work record

A user-defined constraint violation occurs. Display error message and rollback transaction.

END

END
3.6 Implementing COMPANY Triggers Using ORACLE 8

This section explains an implementation of triggers in ORACLE 8. All packages that will be called by triggers are shown in Table 3.3. The general concepts of implementing triggers for a delete, insert and update operation are described in Figure 3.21 - Figure 3.22. These concepts will be applied properly to the EMPLOYEE, DEPARTMENT, LOCATION, PROJECT and WORK_ON tables as described next.

Table 3.3 Packages Used in Implementing Triggers

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJ_ERRORS</td>
<td>Keep all error codes and error messages on integrity constraint violations. It will be called when a violation occurs on tables.</td>
</tr>
<tr>
<td>XXX_TAB_PKG</td>
<td>Store transaction information of a delete, insert or update on records. The function description of this package is explained in Figure 3.20.</td>
</tr>
<tr>
<td>(XXX: DEL_DNO, DEPT, EMP, LOC, PROJ, SSN, WORKS_ON)</td>
<td></td>
</tr>
<tr>
<td>XXXCONS_PKG</td>
<td>Check all integrity constraints depending on tables (DEPARTMENT, EMPLOYEE, LOCATION, PROJECT or WORKS_ON).</td>
</tr>
<tr>
<td>(XXX: DEPT, EMP, LOC, PROJ, WORKS_ON)</td>
<td></td>
</tr>
</tbody>
</table>
PACKAGE XXX_TAB_PKG:
    Declare appropriate temporary tables
    Declare table_count, table_iterator
procedure Clear_table: Clear the table_counter to zero.
procedure Set_iterator: Assign the total number of rows affected by
    a SQL DML statement to the table_iterator variable.
function More_in_table: Check data in table exist.
function Next_rec: Return the data in the temporary table.
procedure Insert_value: Insert data into the temporary table.
Figure 3.21 Implementing Triggers on a Delete Operation

Trigger: A DELETE Operation.

START

XXX: Pertinent to record type
(EMP, DEPT, PROJ, LOC, WORKS_ON)

XXX_TAB_PKG.Clear_table

XXX_TAB_PKG.Insert_value

XXX_CONS_PKG
(check referential integrity constraints)

XXX_TAB_PKG.Set_iterator

XXX_TAB_PKG.More_in_table

no

XXX_TAB_PKG.Next_rec

yes

XXX_CONS_PKG
(check additional referential integrity constraints)
Figure 3.22 Implementing Triggers on an Insert Operation

**Trigger: An Insert Operation.**

```
START
XXX: Pertinent to record type (EMP, DEPT, PROJ, LOC, WORKS_ON)

XXX_TAB_PKG.Clear_table
BEFORE INSERT STATEMENT

XXX_CONS_PKG:
check certain constraints
BEFORE INSERT ROW

-------------
INSERT STATEMENT
-------------

XXX_TAB_PKG.Insert_value
AFTER INSERT ROW

XXX_TAB_PKG.Set_iterator
AFTER INSERT STATEMENT

no

XXX_TAB_PKG.More_in_table

yes

END

XXX_TAB_PKG.Next_rec

XXX_CONS_PKG
(Additional constraints)
```
Trigger: An UPDATE Operation.

The following figure shows an implementation of triggers on a delete operation in the EMPLOYEE and DEPARTMENT tables. These tables are applied similarly to the general DELETE pattern in Figure 3.21. A delete operation in the PROJECT table is implemented differently from those in EMPLOYEE and DEPARTMENT tables (see Figure 3.25).
Figure 3.24 Implementing Triggers on a Delete Operation in the EMPLOYEE and DEPARTMENT Tables

Trigger: A Delete Operation on EMPLOYEE and DEPARTMENT Tables

START  XXX_TAB_PKG:SSN(EMPLOYEE), DEL_DNO(DEPARTMENT)  
        XXX_CONS_PKG: EMP, and DEPT

XXX_TAB_PKG.Clear_table  BEFORE DELETE STATEMENT

DELETE STATEMENT

XXX_TAB_PKG.Insert_value  AFTER DELETE STATEMENT

XXX_CONS_PKG  
(check referential integrity constraints on EMPLOYEE and DEPARTMENT)

XXX_TAB_PKG.Set_iterator

no

XXX_TAB_PKG.More_in_table

yes

XXX_TAB_PKG.Next_rec

EMP: check if a deleted employee is a manager, then set the manager to null and keep the change in the audit table. If the deleted employee works in any project, then delete all work records of this employee.

DEPT: check if a deleted department has employees, then set the department of those employees to null and keep the change in the audit table. Then, delete the locations of deleted department.

EMP: check if a deleted employee is a supervisor, set the supervisor to null and keep the change in the audit table.

DEPT: if a deleted department controls projects, then apply the default department (dno = 10) to those projects and keep the change in audit table.

XXX_CONS_PKG  
(check additional referential integrity constraints)
Figure 3.25 Implementing Triggers on a Delete Operation in the PROJECT Table

Trigger: A Delete Operation on the PROJECT Table.

```
START

PROJ_CONS_PKG: check if a deleted project is referred in the WORKS_ON table, then delete work records in the WORKS_ON table and keep the information of deleted project in audit table.

BEFORE DELETE ROW

DELETE STATEMENT

END
```

For an insert operation, an implementation of triggers in the EMPLOYEE, DEPARTMENT, PROJECT and WORKS_ON tables are described in Figure 3.26. The LOCATION table is explained in Figure 3.27.
Figure 3.26 Implementing Triggers on an Insert Operation in the EMPLOYEE, DEPARTMENT, PROJECT and WORKS_ON Tables

Trigger: An Insert Operation on EMPLOYEE, DEPARTMENT, PROJECT and WORKS_ON Tables.

START

XXX: EMP, DEPT, PROJ, and WORKS_ON

XXX_CONS_PKG
(Primary key, foreign key, not null, check constraint, and user-defined constraint)

BEFORE INSERT ROW
EMPLOYEE: See Figure 3.8
DEPARTMENT: See Figure 3.11
PROJECT: See Figure 3.16
WORKS_ON: See Figure 3.18

INSERT STATEMENT

END
Figure 3.27 Implementing Triggers on an Insert Operation in the LOCATION Table

Trigger: An Insert Operation on LOCATION Table.

START

LOC_TAB_PKG.Clear_table

------

INSERT STATEMENT

LOC_TAB_PKG.Insert_value

------

AFTER INSERT ROW

LOC_TAB_PKG.Set_iterator

------

AFTER INSERT STATEMENT

no

LOC_TAB_PKG.More_in_table

yes

END

LOC_TAB_PKG.Next_rec

LOC_CONS_PKG: check a foreign key constraint on department number and a primary key constraint on both department number and department location.

An implementation of triggers on an update operation in every table is similar as shown in the following figure.
Figure 3.28 Implementing Triggers on an Update Operation in the EMPLOYEE, DEPARTMENT, LOCATION, PROJECT, and WORKS_ON Tables

Trigger: An Update Operation on EMPLOYEE, DEPARTMENT, LOCATION, PROJECT and WORKS_ON Tables.

START

XXX_TAB_PKG.Clear_table

UPDATE STATEMENT

XXX_TAB_PKG.Insert_value

AFTER UPDATE ROW

XXX_TAB_PKG.Set_iterator

AFTER UPDATE STATEMENT

no

XXX_TAB_PKG.More_in_table

yes

XXX_TAB_PKG.Next_rec

EMPELOYEE (see Figure 3.9)
DEPARTMENT (see Figure 3.12)
LOCATION (see Figure 3.14)
PROJECT (see Figure 3.17)
WORKS_ON (see Figure 3.19)

XXX_CONS_PKG: check constraints
(primary key, foreign key, not null, check constraint and user-defined constraint)
3.7 Implementing COMPANY Triggers Using SQL SERVER 7.0

This section describes all triggers implemented in SQL SERVER 7.0. The description of each trigger to enforce integrity constraints is also included. If any violation occurs, these triggers will display an error message and rollback a transaction. An implementation of triggers in SQL SERVER 7.0 is described in Table 3.4.
Table 3.4 Implementing Triggers in SQL SERVER 7.0

<table>
<thead>
<tr>
<th>Trigger</th>
<th>DEL XXX TRG1</th>
<th>INS XXX TRG1</th>
<th>UPD XXX TRG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP</td>
<td>Check a referential integrity constraint when delete an employee. (See Figure 3.7)</td>
<td>Check a primary key on employee number, a not null on first name, last name, birth date and address. A check constraint on employee's gender (male or female). A foreign key on supervisor and department number. The user-defined constraints on the employee’s salary. (See Figure 3.8)</td>
<td>Similar to an insert, but an update will check on a specific column update. An additional verification when update on employee number: if this employee is a supervisor, a manager of department, or employee working in projects, then update them to the new employee number. (See Figure 3.9)</td>
</tr>
<tr>
<td>DEPT</td>
<td>Check a referential integrity constraint when delete a department. (See Figure 3.10)</td>
<td>Check a primary key on department number and foreign key on the manager of department. (See Figure 3.11)</td>
<td>Similar to an insert, but an update checks on a specific column update. An update on department number also checks if the department is referred in the EMPLOYEE, PROJECT, and LOCATION tables, the department number in those tables are also updated. (See Figure 3.12)</td>
</tr>
<tr>
<td>Trigger</td>
<td>DEL</td>
<td>XXX</td>
<td>TRGl</td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td><strong>LOG</strong> Check a foreign key on department number and a primary key on both department number and department location. (See Figure 3.13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PROJ</strong> Check a primary key referential on project number integrity and a foreign key constraint on department when delete a project. (See Figure 3.16)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WORKS ON</strong> Check a foreign key on employee number and project number. Verify a primary key on both of them and check a user-defined constraint: an employee, not a manager, cannot work more than 10 projects. (See Figure 3.18)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UPD</strong> Similar to an insert, but an update will verify on a specific column update, and ensure a primary key constraint. (See Figure 3.14)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INS</strong> Similar to an insert, but an update checks on a particular column update. An extra verification when update a project number: if this project is referred in the WORKS_ON table, then update the project number in WORKS_ON table.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DEL</strong> Similar to an insert, but an update checks on a specific column update, and ensure a primary key constraint if an update occurs on either employee number or project number. (See Figure 3.19)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LOC** Check a foreign key constraint.

**DEL** Check a foreign key constraint on department number and a primary key on department number and a specific column update. An extra verification when delete a department: on department number and a specific column update. (See Figure 3.14)
3.8 Triggers in Auditing Databases

Database auditing is the monitoring and recording of selected user database activities. Information about the event will be stored in the audit trail, which can be used to investigate suspicious activity. For example, to track an unauthorized user who updates data from tables can be done by auditing all connections to the database with the successful and unsuccessful update options in the database.

Auditing database in ORACLE 8 can be performed by the audit statements as follow:

1. Set the AUDIT_TRAIL initialization parameter to enable auditing in the instance.

2. Enable auditing options by the following syntax:

   \[
   \text{AUDIT} \begin{cases} 
   \text{statement|system_priv} \\
   [,\{\text{statement|system_priv}\}] \ldots \\
   \text{[BY user [, user ] ...]} \\
   \text{[BY \{SESSION|ACCESS\}]} \\
   \text{[WHenever \[NOT\] SUCCESSFUL]}
   \end{cases}
   \]

where:

- **Statement** Specifies the SQL statement type or schema-object to audit.
- **System_priv** Specifies the system privilege to audit.
User Indicates to only audit the users in the list.

BY SESSION Causes ORACLE to insert only one record per database object into the audit trial for each session, no matter how many SQL DML statements of the same type are submitted.

BY ACCESS Causes ORACLE to insert a record into the audit trail each time an audited statement is submitted.

WHENEVER Specifies that auditing is to be carried out only on successful or unsuccessful completion of SQL statements.
(The default is both.)

From the above example, auditing database can be done by the following syntax commands:

AUDIT update any table
BY ACCESS;

However, auditing database by using the audit statements records only who made the action, what was done to the data, but not how the data was modified. If there is
a mistake caused by this activity, the data cannot be changed back without recovering it from the backup copy. A trigger, however, makes it possible to track each activity by recording changes to the data on the audit table. The audit table, then, can be used to restore the database when an error occurs.

To illustrate an implementation of triggers for auditing database, the following two triggers will be created for the historical information. The first trigger is used to track changes in the salary of employees. Every time the salary of an employee is updated, an updated record will be kept in the EMP_SAL_AUD table. This table includes the employee number (empno), the old and new salary of the employee (old_sal and new_sal), and the date/time (dateupd) the salary updated. This information can be used to trace the activity and change the data to the last reliable state. The code for generating this trigger is:

```
CREATE TRIGGER EMP_AUD
AFTER UPDATE OF SALARY ON EMPLOYEE
FOR EACH ROW
BEGIN
    INSERT INTO EMP_SAL_AUD
    VALUES(:old.empno, :old.salary, :new.salary,
```
The other trigger is used to track deleted records from the PROJECT table. When a project is finished, this project record might be deleted from the table. The trigger will keep information on the deleted record into the PROJECT_AUD table. This table includes the project number (pnumber), the project name (pname), the department number (dno), the department name (dname), and the department manager (mgrssn). Later, if the company wants to know who was the department manager of the deleted project, the PROJECT_AUD table can be used to retrieve historical information. The code for creating this trigger is:

```
CREATE TRIGGER PROJ_AUD
AFTER DELETE ON PROJECT
DECLARE   mgr department.mgrssn%type;
          dept_dname department.dname%type;
BEGIN
    SELECT d.mgrssn into mgr
    FROM DEPARTMENT d
    WHERE d.dno = :old.dno;
    SELECT d.dname into dept_dname
    FROM DEPARTMENT d
    WHERE d.dno = :old.dno;
```
INSERT INTO PROJECT_AUD
VALUES(:old.pnumber, :old.pname, :old.dno,
    dept_dname, mgr);
END;
A view is a logical table that provides logical windows over table data and it can be manipulated by a DELETE, INSERT or UPDATE operation. A view helps to restrict access to the database because it can display only a selective portion of the database. A view also allows users to make simple queries to retrieve the results from complicated queries. Users can query information from multiple tables without knowing how to write a join statement and they also can create different views of the same data. The syntax for creating a view is:

```
CREATE [OR REPLACE] VIEW view_name
AS sub_query
```

When the view is created from a table, all DELETE, INSERT, and UPDATE statements can be performed. However, if a view is a join of at least two tables, these DML statements are disallowed. To correct this problem, triggers can be used on views. The efficiency of using triggers to maintain views will be demonstrated in the following section.
4.1 View Modification

To modify complex views with the SQL DML statements (DELETE, INSERT, or UPDATE), triggers use three different strategies.

1. Deleting a row from a view:

When the data is deleted from the view, the trigger will delete from the underlying table. If the deleted record in the underlying table is referred to in another child table, the trigger will either delete or modify the child record (See Figure 4.1).

Figure 4.1 Delete a Record from a View

Delete a record from view A:

View: A

<table>
<thead>
<tr>
<th>10</th>
<th>Toy</th>
<th>...</th>
</tr>
</thead>
</table>

Table T1: (Underlying table of view A)

<table>
<thead>
<tr>
<th>10</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

Table C1: (Child table of T1)

(1) delete

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
<th>10</th>
</tr>
</thead>
</table>

Table C2: (Child table of T1)

(2) modify

| ... | ... | 10-null |
2. Inserting a row to a view

When data is inserted into the view, the trigger will insert into the underlying table. In case the inserted record in the view is already in the underlying table, appropriate columns of the underlying tables will be updated (See Figure 4.2).

Figure 4.2 Insert a Record into a View

Insert a record to view A:

View: A

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Electric</td>
<td></td>
</tr>
</tbody>
</table>

Table T1: (Underlying table of view A)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Electric</td>
<td></td>
</tr>
</tbody>
</table>

(1) insert

Table T1: (Underlying table of view A)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Tool</td>
<td>Electric</td>
</tr>
</tbody>
</table>

(2) update

If an inserted record is already in the table, a column value will be updated.
3. Updating a row in a view

When data is updated on the view, the trigger will update the underlying table. If the updated record in the underlying table is referred to by another child table, the trigger will cascade update the update to the child table (See Figure 4.3).

Figure 4.3 Update a Record in the View

Update a record in the view A

<table>
<thead>
<tr>
<th>View: A</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:30 Toy ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table T1: (Underlying table of view A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:30 ... ... ...</td>
</tr>
<tr>
<td>update cascade</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table C1: (Child table of T1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... ... 10:30</td>
</tr>
</tbody>
</table>
4.2 The PROJECT_INFO View

To illustrate the use of trigger in a complex view, the PROJECT_INFO view is created in this project. The PROJECT_INFO view (see Figure 4.4) joins two tables—DEPARTMENT and PROJECT tables. This view tracks project information—project number and project name as well as departments (the department number and department name) that controls the project.

To create the PROJECT_INFO view:

CREATE OR REPLACE VIEW PROJECT_INFO AS
select p.pnumber, p.pname, d.dno, d.dname
from PROJECT p, DEPARTMENT d
where p.dno = d.dno;

Figure 4.4 The PROJECT_INFO View
4.3 A PROJECT_INFO View Modification in ORACLE 8

ORACLE 8 uses a special option, INSTEAD OF, in the trigger command to specify how to update, delete or insert records in the underlying tables when the value in a view is modified. Unlike ordinary triggers, the INSTEAD OF option can be applied to views only. It cannot be applied to a table. Additionally, this option is not allowed in the statement-level trigger. It can be activated only at the row-level trigger.

By implementing triggers to the PROJECT_INFO view, a DML statement will not be executed on the view but it will be executed directly on the underlying tables--PROJECT and DEPARTMENT tables. Any integrity constraint violation that may occur when manipulating the PROJECT_INFO view will be verified on these tables as well.

Each SQL statement DELETE, INSERT and UPDATE on the PROJECT_INFO view can be applied as follows.

1. Delete a record from the view

   When a delete operation occurs in the PROJECT_INFO view, the record will be deleted from the PROJECT table instead. If the deleted record in the PROJECT table is referred to from another table
(WORKS_ON), the triggers will delete cascade to WORKS_ON table as shown below.

Figure 4.5 A Delete Operation on the PROJECT_INFO View

```
DELETE FROM PROJECT_INFO
WHERE pnumber = 3 and dno = 4;
```

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>DNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ProductX</td>
<td>2</td>
<td>Administration</td>
</tr>
<tr>
<td>3</td>
<td>ProductY</td>
<td>4</td>
<td>Accounting</td>
</tr>
<tr>
<td>4</td>
<td>ProductZ</td>
<td>4</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>DNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ProductX</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ProductY</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>ProductZ</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>PNUMBER</th>
<th>HOURS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100055001</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>100055002</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>100043003</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>100052004</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

2. Insert a record into a view

An insert of a new record into the PROJECT_INFO view implies inserting a new project for an existing department as well as a new project for a new department.

An insert of a new record for an existing department requires the project number, project name and department number. The name of department is
allowed to be omitted since the inserted department exists. However, if the name of an inserted department is given, it must be the same as that of an existing one. If no violation occurs, the information on the project will be inserted in the PROJECT table as shown in Figure 4.6.

Figure 4.6 An Insert of a New Project Record for an Existing Department into the PROJECT_INFO View

<table>
<thead>
<tr>
<th>Case 1: The name of department is omitted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO PROJECT_INFO (pnumber, pname, dno)</td>
</tr>
<tr>
<td>VALUES (5, 'ProductM', 4);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 2: The name of department is given.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO PROJECT_INFO (pnumber, pname, dno, dname)</td>
</tr>
<tr>
<td>VALUES (5, 'ProductM', 4, 'Accounting');</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROJECT_INFO:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNUMBER</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

An insert of a new project for a new department requires all column values. The new project information will be inserted into the PROJECT table, and the new department will be inserted into the DEPARTMENT table. A
default manager (mgrssn = 999999999) will also be assigned for a new department as shown in the following figure.

Figure 4.7 An Insert of a New Project and Department Record into the PROJECT_INFO View

```
INSERT INTO PROJECT_INFO
VALUES (5, 'ProductM', 5, 'Marketing');
```

**PROJECT_INFO:**

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>DNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ProductX</td>
<td>2</td>
<td>Administration</td>
</tr>
<tr>
<td>3</td>
<td>ProductY</td>
<td>4</td>
<td>Accounting</td>
</tr>
<tr>
<td>4</td>
<td>ProductZ</td>
<td>4</td>
<td>Accounting</td>
</tr>
<tr>
<td>5</td>
<td>ProductM</td>
<td>5</td>
<td>Marketing</td>
</tr>
</tbody>
</table>

**PROJECT:**

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>DNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ProductX</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ProductY</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>ProductZ</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>ProductM</td>
<td>5</td>
</tr>
</tbody>
</table>

**DEPARTMENT:**

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>1112222333</td>
</tr>
<tr>
<td>2</td>
<td>Administration</td>
<td>3334445555</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>5556667777</td>
</tr>
<tr>
<td>4</td>
<td>Accounting</td>
<td>123456789</td>
</tr>
<tr>
<td>5</td>
<td>Marketing</td>
<td>9999999999</td>
</tr>
</tbody>
</table>

3. Update a record in a view

When the project information in the PROJECT_INFO view is updated, it will be updated at the PROJECT table. An update on the PROJECT_INFO view can take one of the following forms:
- Change department number to an existing department. This implies that the control of this project is being transferred to another department.

- Change project name corresponding to an existing project number. This project name cannot be one of the existing project names.

- Change the project number for an existing project controlled by the department to a new project. If this project is referred in the child table (WORKS_ON), the project number in the WORKS_ON table is also updated (see Figure 4.8).

An update operation is disallowed on the name of department since that department may also control other projects.
4.4 Implementing the PROJECT_INFO View in ORACLE 8

The following figures shows the flowcharts of implementing the PROJECT_INFO view on delete, insert and update operation (Figure 4.9 - Figure 4.11). The source code can be found in Appendix C.
DELETE FROM PROJECT_INFO VIEW:

START
Delete PROJECT_INFO VIEW with dno = x and pnumber = y

Does this record exist in the view?

no
Delete cannot be done.
END

yes
Can delete this record, but need to check for integrity constraints

Does this department exist in the DEPARTMENT table [dno == dno in DEPARTMENT table]

no
Delete cannot be done.
END

yes

Does this project exist in the PROJECT table [pnumber == pnumber in PROJECT table]

no
Delete cannot be done.
END

yes
Delete this project record from the PROJECT table. If the deleted project is referred in the work records, all work records of this project are also deleted.

END

END
INSERT PROJECT_INFO VIEW:

START

Can insert this record, but need to check for integrity constraints

no

Does this department exist in the DEPARTMENT table
[dno == dno in DEPARTMENT table]

yes

Insert the department number, department name and a default manager of department into the DEPARTMENT table

no

Is the name of this department different from that of an existing department [dname <> an existing dname]

yes

Is it not null [dname is not null]

no

END

yes

A primary key violation on the PROJECT table. Display error message and rollback transaction

END

The department exists, not allow to insert a different name. Display error message and rollback transaction

no

Does this project exist in the PROJECT table [pnumber == pnumber in PROJECT table]

yes

A unique key violation on the PROJECT table. Display error message and rollback transaction

no

Does the name of project exist in the PROJECT table [pname == pname in other records]

yes

Insert the project number, project name and department number into the PROJECT table.

no

END

END

END

87
Figure 4.11 Update on the PROJECT_INFO View

UPDATE PROJECT_INFO VIEW:

START

Does this record exist in the view

yes Can update this record, but need to check integrity constraints

no

Update cannot be done

END

Is the project number updated?

no

[update(pnumber)]

yes

Is it updated to an existing project number?

no [pnumber == pnumber in the PROJECT table]

yes [pnumber updated]

[A primary key violation. Display error message and rollback transaction.]

Update project number in the PROJECT table. If it is also referred in work records, then update the project number of those work records.

END

Is the project name updated?

no

[update(pname)]

yes

Is it updated to an existing name?

no [pname == pname in other records]

yes [pname updated]

[update(pname)]

[Update the project's name in the PROJECT table to the new name.]

[End]

Is the department number updated?

no [update(dno)]

yes

Is it updated to an existing department?

no [dno == dno in DEPARTMENT]

yes [dno updated]

[The department does not exist. Display error message and rollback transaction.]

[Update the department of the project to the new department number.]

[End]

The department name is also used in other projects. Not allow to update. Display error message and rollback transaction.

[End]

[End]

[End]
4.5 A PROJECT_INFO View Modification in SQL SERVER 7.0

Since PROJECT_INFO view is a join of two tables, DEPARTMENT and PROJECT, SQL SERVER 7.0 will not allow explicit DELETE, INSERT, or UPDATE on the view. A manipulation on the PROJECT_INFO view in SQL SERVER 7.0 is done by issuing operation directly on the base tables and not on the views (see Table 4.1).

Table 4.1 A PROJECT_INFO View Modification in SQL SERVER 7.0 and ORACLE 8

<table>
<thead>
<tr>
<th>SQL SERVER 7.0</th>
<th>ORACLE 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete from PROJECT</td>
<td>delete from PROJECT_INFO</td>
</tr>
<tr>
<td>where pnumber = 3 and dno = 4;</td>
<td>where pnumber = 3 and dno = 4;</td>
</tr>
<tr>
<td>1. Insert to the DEPARTMENT table</td>
<td>insert into PROJECT_INFO</td>
</tr>
<tr>
<td>insert into department</td>
<td>values (5, 'ProductM', 5, 'Marketing');</td>
</tr>
<tr>
<td>values(5, 'Marketing');</td>
<td></td>
</tr>
<tr>
<td>2. Insert to the PROJECT table</td>
<td>insert into project</td>
</tr>
<tr>
<td>insert into project</td>
<td>values(5, 'Marketing', 5);</td>
</tr>
<tr>
<td>values(5, 'Marketing', 5);</td>
<td></td>
</tr>
<tr>
<td>update PROJECT</td>
<td>update PROJECT_INFO</td>
</tr>
<tr>
<td>set pnumber = 6</td>
<td>set pnumber = 6</td>
</tr>
<tr>
<td>where pnumber = 4 and dno = 4;</td>
<td>where pnumber = 4 and dno = 4;</td>
</tr>
</tbody>
</table>
CHAPTER FIVE: ANALYSIS

In this chapter, the declarative and trigger methods will be compared based on ease of implementation, flexibility, execution time, limitation, view management, and database audit.

5.1 Comparison Between Declarative and Trigger Methods

There are both advantages and disadvantages in the declarative and trigger methods. The declarative method is easier and faster than the trigger method. However, the trigger method provides more flexibility than the declarative method does. Table 5.1 shows which comparison criterion is satisfied by the declarative and trigger methods. A detailed description will be discussed in the following sections.
Table 5.1 Declarative Versus Trigger Methods

<table>
<thead>
<tr>
<th>Comparison Criterion</th>
<th>Declarative method</th>
<th>Trigger method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not null constraint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Unique key constraint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Primary key constraint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Foreign key constraint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Check constraint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Verify the existing data</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Ease of use</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Execution time</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Flexibility</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

5.1.1 Implementing Integrity Constraints

Both declarative and trigger methods can be used to correct integrity constraints when a violation occurs on unique key, primary key, foreign key, not null, and check constraint. However, the trigger method requires more complicated SQL programs to be written, while the declarative method declares all of these constraints using a simple SQL command (see Table 5.2).

5.1.2 Verifying the Existing Data

When implementing integrity constraints with the declarative method, all existing data in the database will
be checked for violations. An alert message will notify the user of the invalid data. This system alert message makes it easier to detect and correct the invalid data. The integrity constraints will not be allowed to activate, until all invalid data are corrected.

In the trigger method, however, only newly entered data are verified. Any existing data will not be checked for violations. If the existing data is invalid, an alert message will not be given. This makes it harder to locate inaccurate data. Therefore, all existing data must be carefully validated before implementing triggers.

5.1.3 Ease of Use

The implementation process of the declarative method is much easier than that of trigger method. The integrity constraints are simply added during the table definition. After an integrity constraint is declared, it is easy to modify the constraint at anytime. Furthermore, both declaration and modification of integrity constraints can be defined using only one simple SQL statement (CREATE TABLE, or ALTER TABLE ... ADD CONSTRAINT ...).

The implementation process of triggers is more complicated. Triggers cannot be implemented during the table definition process. They must be generated only after the table already exists. Triggers will be created by using
complex SQL statements, which are difficult to understand and implement, SQL expertise is required. Coding, testing, and debugging processes of triggers are also time consuming. The following table, Table 5.2, shows the number of statements used in both declarative and trigger methods in the EMPLOYEE tables.

Table 5.2 Statements Used to Enforce Constraints in the Declarative and Trigger Methods—EMPLOYEE Table

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Declarative Method</th>
<th>Trigger Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Null</td>
<td>1</td>
<td>92</td>
</tr>
<tr>
<td>Unique Key</td>
<td>1</td>
<td>112</td>
</tr>
<tr>
<td>Primary Key</td>
<td>1</td>
<td>132</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>1</td>
<td>98</td>
</tr>
<tr>
<td>Foreign Key</td>
<td>1</td>
<td>198</td>
</tr>
</tbody>
</table>

From the above table, the trigger method requires a higher number of statements than the declarative method does (only one statement). In declarative method, when a constraint is declared, the system automatically enforces constraints from insert, update and delete operations. In the trigger method, SQL statements must be implemented to handle any possible violations that might occur from all of these operations.
5.1.4 Execution Time

Table 5.3 shows that the average execution times of the declarative method are less than that of trigger method. The execution time is measured by taking the difference between the recorded system time before using an operation and the recorded system time after the operation is run. After the same number of records are inserted into both tables ten times, the average execution time is calculated by adding all execution times and divided by ten. The following is the information on the EMPLOYEE table that is used for testing the performance.

Employee Table:

Number of fields: 11
Length of a record: 149 bytes

Table 5.3 The Average Execution Time Using Declarative and Trigger Methods for Insert Operation

<table>
<thead>
<tr>
<th>Sizes of records (Kb)</th>
<th>Average Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Declarative Method</td>
</tr>
<tr>
<td>0.72</td>
<td>0.2</td>
</tr>
<tr>
<td>2.18</td>
<td>0.5</td>
</tr>
<tr>
<td>3.63</td>
<td>1.0</td>
</tr>
<tr>
<td>5.09</td>
<td>1.8</td>
</tr>
</tbody>
</table>
Table 5.4 and Table 5.5 show the execution time used for delete and update operations using declarative and trigger methods. The measurements of the execution time in Table 5.4 and Table 5.5 are similar to that of insert operation by taking the difference between the recorded system time before using an operation and the recorded system time after executing an operation.

Table 5.4 The Execution Time Using Declarative and Trigger Methods for Delete Operation

<table>
<thead>
<tr>
<th>Query</th>
<th>Number of Employee Records</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Declarative Method</td>
<td>Trigger Method</td>
</tr>
<tr>
<td>case 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>case 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>case 3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>case 4</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Case description for Table 5.4 and Table 5.5:
Case 1: Delete/Update an employee who also works on the projects
Case 2: Delete/Update an employee who is a supervisor and a manager of department, and also works on the projects
Case 3: Delete/Update employees of a supervisor
Case 4: Delete/Update employees on a department
Table 5.5 The Execution Time Using Declarative and Trigger Methods for Update Operation

<table>
<thead>
<tr>
<th>Query</th>
<th>Number of Employee Records</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Declarative Method</td>
</tr>
<tr>
<td>case 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>case 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>case 3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>case 4</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

From Table 5.3, Table 5.4 and Table 5.5, the execution times used for insert, delete and update operations in the declarative method are faster than the execution times used in the trigger method. In the trigger method, the time involved in running a trigger is spent mostly in referencing other tables, which can be either in memory or on the database device. This means when a trigger is fired, not only does the database system execute that operation, but it also takes time to compile other SQL statements that are in the trigger, as well. If those SQL statements refer to other tables not in cache memory, it will take extra time to read data from the database device. The location of other tables referenced by the trigger determines the amount of time the operation requires. The more the SQL
statements the trigger method has, the longer the time it uses to compile and execute.

5.1.5 Flexibility

In the declarative method, all integrity constraints are corrected by the database system. The repair strategies will be automatically defined by the system. These repair strategies are unchangeable. Although, the system will generate standard error messages, these are hard to understand, and might not provide enough information to handle problems (See Figure 5.1).

Figure 5.1 Error Message Provided by System

The powerful advantage of the triggers is flexibility. When triggers are implemented to correct an integrity violation, the repair strategies are adaptable depending on a situation. Also, triggers will display a more understandable error message (See Figure 5.2) to the users.
instead of using the system error message that may give an unclear explanation for the problems.

Figure 5.2 Error Message Provided by the Trigger Method

5.1.6 Complex User-defined Constraints

In declarative method, the user-defined constraint normally can be verified by using the CHECK constraint statement. However, the CHECK statement sometimes cannot correct user-defined constraints when the constraint needs to reference data in another table. The CHECK constraint is defined for fields within one table only.

The trigger is an excellent method to implement the user-defined constraint that cannot be modified by the CHECK constraint statement. It can implement complex user-defined constraints and perform the same functionality as the CHECK statement. An example of complex user-defined constraint implemented by trigger in this project is:
The salary of employee who is not a manager cannot get a raise in salary more than 15%.

This complex user-defined constraint needs to check the information of two tables--EMPLOYEE and DEPARTMENT tables. It also needs to check the change in employee's salary--the difference between the new and the old salary. This constraint cannot be performed by using the CHECK constraint.

5.1.7 View Maintenance

A view can be manipulated by SQL DML (DELETE, INSERT, and UPDATE) statements if the view derives data from only one table. When the view is more complex--a join between two tables or more--a modification on the view is not permitted. By using the trigger, the modification of a join-table view is acceptable. Triggers will perform SQL DML operation directly on the underlying tables instead of the view.

5.1.8 Database Auditing

Any database activity, such as authentication and manipulation, can be audited by using the audit statements. However, an audit statement will keep only the event information. It will not record changes of the column values. This means auditing the database cannot track the modification of the column values. Trigger is an efficient
tool used to audit the alteration of the column values. The old and/or new values are stored every time the data is modified. This makes it easier to track any change of the data.

5.2 Comparison Between Triggers in ORACLE 8 and SQL SERVER 7.0

Triggers in both ORACLE 8 and SQL SERVER 7.0 systems are used to enforce an integrity constraint when a violation occurs. The trigger features in both systems are compared in Table 5.6 and the description follows.
Table 5.6 Triggers in ORACLE 8 Versus Triggers in SQL SERVER 7.0

<table>
<thead>
<tr>
<th>Criterion</th>
<th>ORACLE 8</th>
<th>SQL SERVER 7.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of triggers per table</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of trigger levels</td>
<td>Unlimited</td>
<td>32 levels</td>
</tr>
<tr>
<td></td>
<td>(Default=32)</td>
<td></td>
</tr>
<tr>
<td>Statement-level triggers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Row-level triggers</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Trigger executed BEFORE INSERT, UPDATE, DELETE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Trigger executed AFTER INSERT, UPDATE, DELETE</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Combination of triggers</td>
<td>12 types</td>
<td>3 types</td>
</tr>
<tr>
<td>Referring to new values in an INSERT trigger</td>
<td>:new.column</td>
<td>Inserted. column</td>
</tr>
<tr>
<td>Referring to old values in an DELETE trigger</td>
<td>:old.column</td>
<td>Deleted.column</td>
</tr>
<tr>
<td>View management</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>(Instead of)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declarative Integrity Constraints</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Complex business rules</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Verify the existing data</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
5.2.1 Number of Triggers

The number of triggers in both ORACLE 8 and SQL SERVER 7.0 is unlimited. Both of them allow creating triggers on different types of DML statements—DELETE, INSERT or UPDATE. For each DML statement, there are an unlimited number of triggers used. Many DML statements are able to share the same trigger.

A trigger can activate any other trigger, which in-turn may activate other triggers. For example, Trigger A activates Trigger B, and then Trigger B may activate Trigger C, and so on. The number of trigger levels in ORACLE 8 is flexible. The default value is 32 levels, and it can be changed by using OPEN_CURSOR parameter. The level of triggers in SQL SERVER 7.0, however, is limited to 32 levels.

5.2.2 Trigger Level

ORACLE 8’s triggers can be implemented in both row-level—fired once for each row affected by the SQL statement—and a statement-level—fired once per statement. All rows affected by the SQL statement will be verified by triggers.

In SQL SERVER 7.0, triggers can only be statement-level. This means if more than one row is affected by the SQL statement, a trigger will be fired only on the last
row. To implement the multiple rows, the trigger has to perform a delete, insert or update statement in a loop.

5.2.3 Trigger Timing

There are two types of trigger timing--BEFORE or AFTER. The trigger timing indicates when the trigger action will be executed. BEFORE and AFTER triggers can be specified in ORACLE 8 while triggers in SQL SERVER 7.0 is limited to the AFTER trigger timing only.

5.2.4 Trigger Combination

ORACLE 8 can create triggers in both two level triggers and two trigger timings for each SQL DML statement (DELETE/INSERT/UPDATE). Therefore, ORACLE 8 can provide up to twelve trigger combination types (See Figure 5.3). SQL SERVER 7.0, however, can only have statement-level trigger and AFTER trigger timing for each DML statement. The trigger combinations in SQL SERVER 7.0 are limited to only three types.
5.2.5 Referencing Data

Both systems use aliases to refer to a record that is inserted or deleted. In SQL SERVER 7.0, these aliases refer to special tables—the INSERTED and DELETED tables—, which are created when executing triggers. The INSERTED table contains a copy of every row to be added or updated (new value) and the DELETED table contains a copy of every row to be deleted or updated (old value).

ORACLE 8 uses the aliases :new and :old records to refer to an inserted or deleted data. The :new keeps the value of a new record after an insert or update statement.
The :old keeps the value of an old record prior to change from an update statement or the value of the deleted record on a delete statement.

5.2.6 Maintenance of Complicated Views

ORACLE 8 has an INSTEAD OF trigger option to manage complicated views. The DML statements will be automatically executed on the underlying tables instead of on the view. SQL SERVER 7.0, however, does not provide this feature. The DML statements must be executed manually on the underlying tables.

Both ORACLE 8 and SQL SERVER 7.0 can implement triggers to correct integrity constraint violations, which include the user-defined constraints. Triggers will apply data only after they are created and enabled. Additionally, both of them can also create triggers without limitation in the number of triggers. They can refer to the old and new value when deleting, inserting, or updating a record. The difference is that ORACLE 8 provides a row-level trigger, an AFTER trigger timing, a higher number of combination triggers, and the INSTEAD OF trigger option for the complex view while SQL SERVER 7.0 does not.
CHAPTER SIX: CONCLUSION AND FUTURE WORK

6.1 Conclusion

This project presented the efficient mechanism to implement database integrity constraints and view managements. It compared and showed strengths and weaknesses of both conventional and active database systems. The model databases also provided comprehensive view of how to employ the active rules in database systems.

When compared with the conventional database system, the active database system provides complex user-defined constraints, complicated view managements and more detailed database auditing systems. However, it requires higher skill in programming, more time to develop, higher system resources, and longer execution time. Since there are both advantages and disadvantages, the developers should consider the mixed database system to obtain the highest possible benefit.

Finally, this project strengthened my knowledge in the database field, including the design, development, implementation, manipulation, and maintenance of commercial database systems. It also provided an in-depth understanding of programming concept through SQL, PL/SQL, and T-SQL languages. All of these skills are beneficial for my career in the future.
6.2 Future Work

Triggers normally correct integrity constraint violations automatically without interacting with the end-users. Currently, when a supervisor EMPLOYEE is deleted from the EMPLOYEE table, the supervisor SSN of the supervisees are set to null. In the future, it would be nice to have an interface to prompt the user on the various options to deal with the supervisor value. Such options might include (1) prompting for a new supervisor value, (2) allowing a default value or (3) setting the value to null.

To overcome this problem, a user interface should be implemented as a front-end application. Users will interact with this application and makes a decision on the change of data. This project, however, deals with the back-end application only. The front-end application and the user interface, which are not developed in this project, can be implemented in the future.

In addition, if a user makes a mistake by deleting a wrong employee, the work records of this employee will also be deleted. Although these records can be recovered from the backup, the recover process is inconvenient and time consuming.

Trigger provides a more efficient way to overcome this problem. When a user updates or deletes any record, the
changes will be kept into an audit table, which will be used to recover data when the user makes a mistake. An implementation of triggers for storing this information can be developed in the future.
APPENDIX A: TABLES AND VIEWS

The COMPANY1 Database in SQL SERVER 7.0

DEPARTMENT:
CREATE TABLE department
(dno int,
dname varchar(40),
mgrssn varchar(11));

EMPLOYEE:
CREATE TABLE employee
(empno varchar(11),
fname varchar(20),
mint char(1),
lname varchar(25),
bdate datetime,
address varchar(40),
tel varchar(20),
sex char(1),
salary money,
ssn varchar(11),
dno int);

LOCATION:
CREATE TABLE location
(dno int,
dlocation varchar(30));

PROJECT:
CREATE TABLE project
(pnumber int,
pname varchar(30),
dno int);

WORKS_ON:
CREATE TABLE works_on
(empno varchar(11),
pnumber int,
hours real);

DEPARTMENT_AUD:
CREATE TABLE department_aud
(dno int,
dname varchar(40),
employee varchar(11),
del_date datetime);

DEPT_OF_PROJ_AUD:
CREATE TABLE dept_of_proj_aud
(pnumber int,
pname varchar(30),
dno int,
dname varchar(40),
del_date datetime);
EMP_SAL_AUD:
CREATE TABLE emp_sal_aud
(empno varchar(11),
old_sal money,
new_sal money,
dateupd datetime);

MANAGER_AUD:
CREATE TABLE manager_aud
(manager varchar(11),
dno int,
dname varchar(40),
del_date datetime);

PROJECT_AUD:
CREATE TABLE project_aud
(pnumber int,
pname varchar(30),
dno int,
dname varchar(40),
mgrssn varchar(11),
del_date datetime);

SUPERVISOR_AUD:
CREATE TABLE supervisor_aud
(supervisor varchar(11),
supervisee varchar(11),
del_date datetime);

PROJECT_INFO VIEW:
Create or replace view project_info as
SELECT p.pnumber, p.pname, d.dno, d.dname
FROM project p, department d
WHERE p.dno = d.dno
The COMPANY2 Database in SQL SERVER 7.0

DEPARTMENT:
CREATE TABLE department
  (dno int NOT NULL,
   dname varchar(40) NOT NULL,
   mgrssn varchar(11));

EMPLOYEE:
CREATE TABLE employee
  (empno varchar(11) NOT NULL,
   fname varchar(20) NOT NULL,
   mint char(1),
   lname varchar(25) NOT NULL,
   bdate datetime NOT NULL,
   address varchar(40) NOT NULL,
   tel varchar(20),
   sex char(1),
   salary money NOT NULL,
   ssn varchar(11),
   dno int);

LOCATION:
CREATE TABLE location
  (dno int NOT NULL,
   dlocation varchar(30) NOT NULL);

PROJECT:
CREATE TABLE project
  (pnumber int NOT NULL,
   pname varchar(30) NOT NULL,
   dno int NOT NULL);

WORKS_ON:
CREATE TABLE works_on
  (empno varchar(11) NOT NULL,
   pnumber int NOT NULL,
   hours real);

EMP_SAL_AUD:
CREATE TABLE emp_sal_aud
  (empno varchar(11),
   old_sal money,
   new_sal money,
   dateupd datetime);
Constraints:

ALTER TABLE employee
ADD CONSTRAINT emp_empno_pk PRIMARY KEY (empno);
ALTER TABLE employee
ADD CONSTRAINT emp_sex_ck CHECK (sex in ('F', 'M', 'f', 'm'));
ALTER TABLE employee
ADD CONSTRAINT emp_salary_ck CHECK (salary >= 0);
ALTER TABLE employee
ADD CONSTRAINT emp_ssn_fk FOREIGN KEY (ssn)
    REFERENCES employee (empno);
ALTER TABLE department
ADD CONSTRAINT dept_dno_pk PRIMARY KEY (dno);
ALTER TABLE department
ADD CONSTRAINT dept_dname_uk UNIQUE (dname);
ALTER TABLE employee
ADD CONSTRAINT emp_dno_fk FOREIGN KEY (dno)
    REFERENCES department (dno);
ALTER TABLE department
ADD CONSTRAINT dept_mgrssn_fk FOREIGN KEY (mgrssn)
    REFERENCES employee (empno);
ALTER TABLE works_on
ADD CONSTRAINT works_empno_fk FOREIGN KEY (empno)
    REFERENCES employee (empno);
ALTER TABLE project
ADD CONSTRAINT proj_pnum_pk PRIMARY KEY (pnumber);
ALTER TABLE project
ADD CONSTRAINT proj_pname_uk UNIQUE (pname);
ALTER TABLE works_on
ADD CONSTRAINT works_pnum_fk FOREIGN KEY (pnumber)
    REFERENCES project (pnumber);
ALTER TABLE works_on
ADD CONSTRAINT works_emp_pnum_pk PRIMARY KEY (empno, pnumber);
ALTER TABLE works_on
ADD CONSTRAINT works_hours_ck CHECK (hours >= 0);
ALTER TABLE project
ADD CONSTRAINT proj_dno_fk FOREIGN KEY (dno)
    REFERENCES department (dno);
ALTER TABLE location
ADD CONSTRAINT loc_dno_fk FOREIGN KEY (dno)
    REFERENCES department (dno);
ALTER TABLE location
ADD CONSTRAINT loc_dno_dloc_pk PRIMARY KEY (dno, dlocation);
The COMPANY1 Database in ORACLE 8

DEPARTMENT:
CREATE TABLE department
(dno NUMBER(4),
dname VARCHAR2(40),
mgrssn VARCHAR2(11));

EMPLOYEE:
CREATE TABLE employee
(empno VARCHAR2(11),
fname VARCHAR2(20),
mint CHAR(1),
 lname VARCHAR2(25),
 bdate DATE,
 address VARCHAR2(40),
tel VARCHAR2(20),
sex CHAR(1),
salary NUMBER(8),
ssn VARCHAR2(11),
dno NUMBER(4));

LOCATION:
CREATE TABLE location
(dno NUMBER(4),
dlocation VARCHAR2(30));

PROJECT:
CREATE TABLE project
(pnumber NUMBER(4),
pname VARCHAR2(30),
dno NUMBER(4));

WORKS_ON:
CREATE TABLE works_on
(empno VARCHAR2(11),
pnumber NUMBER(4),
hours NUMBER(5,2));

DEPARTMENT_AUD:
CREATE TABLE department_aud
(dno NUMBER(4),
dname varchar2(40),
 employee varchar2(11),
del date date,
del_time varchar2(11));

DEPT_OF_PROJ_AUD:
CREATE TABLE dept_of_proj_aud
(pnumber NUMBER(4),
pname VARCHAR2(30),
dno NUMBER(4),
dname VARCHAR2(40),
del_date DATE,
del_time VARCHAR2(10));
EMP_SAL_AUD:
CREATE TABLE emp_sal_aud
(empno VARCHAR2(11),
 old_sal NUMBER(8),
 new_sal NUMBER(8),
 dateupd DATE,
 timeupd VARCHAR2(10));

MANAGER_AUD:
CREATE TABLE manager_aud
(manager VARCHAR2(11),
 dno NUMBER(4),
 dname VARCHAR2(40),
 del_date DATE,
 del_time VARCHAR2(10));

PROJECT_AUD:
CREATE TABLE project_aud
(pnumber NUMBER(4),
 pname VARCHAR2(30),
 dno NUMBER(4),
 dname VARCHAR2(40),
 mgrssn VARCHAR2(11),
 del_date DATE,
 del_time VARCHAR2(10));

SUPERVISOR_AUD:
CREATE TABLE supervisor_aud
(supervisor VARCHAR2(11),
 supervisee VARCHAR2(11),
 del_date DATE,
 del_time VARCHAR2(10));

PROJECT_INFO VIEW:
create or replace view project_info as
select d.dno, d.dname, p.pnumber, p.pname
from department d, project p
where d.dno = p.dno;
The COMPANY2 Database in ORACLE 8

DEPARTMENT:
CREATE TABLE department
(dno NUMBER(4) NOT NULL,
dname VARCHAR2(40) NOT NULL,
mgrssn VARCHAR2(11));

EMPLOYEE:
CREATE TABLE employee
(empno VARCHAR2(11) NOT NULL,
fname VARCHAR2(20) NOT NULL,
mint CHAR(1),
lnname VARCHAR2(25) NOT NULL,
bdate DATE NOT NULL,
address VARCHAR2(40) NOT NULL,
tel VARCHAR2(20),
sex CHAR(1),
salary NUMBER(8) NOT NULL,
ssn VARCHAR2(11),
dno NUMBER(4));

LOCATION:
CREATE TABLE location
(dno NUMBER(4) NOT NULL,
dlocation VARCHAR2(30) NOT NULL);

PROJECT:
CREATE TABLE project
(pnumber NUMBER(4) NOT NULL,
pname VARCHAR2(30) NOT NULL,
dno NUMBER(4));

WORKS_ON:
CREATE TABLE works_on
(empno VARCHAR2(11) NOT NULL,
pnumber NUMBER(4) NOT NULL,
hours NUMBER(5,2));

EMP_SAL_AUD:
CREATE TABLE emp_sal_aud
(empno VARCHAR2(11),
old_sal NUMBER(8),
new_sal NUMBER(8),
dateupd DATE,
timeupd VARCHAR2(10));
Constraints:

ALTER TABLE employee
ADD CONSTRAINT emp_empno_pk PRIMARY KEY (empno);
ALTER TABLE employee
ADD CONSTRAINT emp_sex_ck CHECK (sex in ('F', 'M', 'f', 'm'));
ALTER TABLE employee
ADD CONSTRAINT emp_salary_ck CHECK (salary >= 0);
ALTER TABLE employee
ADD CONSTRAINT emp_ssn_fk FOREIGN KEY (ssn)
    REFERENCES employee (empno) ON DELETE CASCADE;
ALTER TABLE department
ADD CONSTRAINT dept_dno_pk PRIMARY KEY (dno);
ALTER TABLE department
ADD CONSTRAINT dept_dname_uk UNIQUE (dname);
ALTER TABLE employee
ADD CONSTRAINT emp_dno_fk FOREIGN KEY (dno)
    REFERENCES department (dno) ON DELETE CASCADE;
ALTER TABLE department
ADD CONSTRAINT dept_mgrssn_fk FOREIGN KEY (mgrssn)
    REFERENCES employee (empno) ON DELETE CASCADE;
ALTER TABLE works_on
ADD CONSTRAINT works_empno_fk FOREIGN KEY (empno)
    REFERENCES employee (empno) ON DELETE CASCADE;
ALTER TABLE project
ADD CONSTRAINT proj_pnum_pk PRIMARY KEY (pnumber);
ALTER TABLE project
ADD CONSTRAINT proj_pname_uk UNIQUE (pname);
ALTER TABLE works_on
ADD CONSTRAINT works_pnum_fk FOREIGN KEY (pnumber)
    REFERENCES project (pnumber) ON DELETE CASCADE;
ALTER TABLE works_on
ADD CONSTRAINT works_emp_pnum_pk PRIMARY KEY (empno, pnumber);
ALTER TABLE works_on
ADD CONSTRAINT works_hours_ck CHECK (hours >= 0);
ALTER TABLE project
ADD CONSTRAINT proj_dno_fk FOREIGN KEY (dno)
    REFERENCES department (dno) ON DELETE CASCADE;
ALTER TABLE location
ADD CONSTRAINT loc_dno_fk FOREIGN KEY (dno)
    REFERENCES department (dno) ON DELETE CASCADE;
ALTER TABLE location
ADD CONSTRAINT loc_dno_dloc_pk PRIMARY KEY (dno, dlocation);
APPENDIX B: LOADING DATA INTO DATABASES

DEPARTMENT:

insert into department(dno, dname, mgrssn)
values (1, 'Research', '111222333');

insert into department(dno, dname, mgrssn)
values (2, 'Administration', '333444555');

insert into department(dno, dname, mgrssn)
values (3, 'Headquarters', '555666777');

insert into department(dno, dname, mgrssn)
values (4, 'Environment', '777888999');

insert into department(dno, dname, mgrssn)
values (10, 'Temporary', '999999999');

EMPLOYEE:

insert into employee
values ('123456789', 'John', 'B', 'Smith', '1-may-1967', '5026 Hollywood Blvd., Los Angeles, CA', '323-660-8999', 'M', 31000, '111000111', 1);

insert into employee

insert into employee
values ('123456786', 'Jennifer', 'L', 'Wallace', '3-jun-1965', '10801 Sherman way, Sun Valley CA', '818-764-9235', 'F', 40000, '111000111', 1);

insert into employee
values ('123456785', 'Armad', 'A', 'Joyee', '4-july-1961', '5108 Fullerton Rd, Rowland Height, CA', '323-660-8999', 'M', 36000, '111000111', 1);

insert into employee
values ('123456784', 'Kanda', NULL, 'Dundee', '5-aug-1971', '1010 Fullerton Rd, Rowland Height, CA', '626-660-8799', 'F', 37000, '222000222', 2);

insert into employee
values ('123456783', 'Suchat', NULL, 'Tim', '6-dec-1970', '5108 Fullerton Rd, Rowland Height, CA', '714-460-8999', 'F', 33000, '333000333', 3);

insert into employee
values ('123456782', 'John', 'W', 'Lee', '7-jan-1980', '456 Pacific Coast HWY, Long Beach, CA', '562-591-1099', 'M', 31000, '111000111', 1);

insert into employee
values ('123456781', 'Jane', 'B', 'Kong', '8-feb-1979', '555 Pacific Coast HWY, Long Beach, CA', '562-523-3399', 'F', 35000, '222000222', 2);
insert into employee
values ('123456780', 'Ramesh', 'L', 'English', '9-apr-1978', '5126 Hollywood Blvd., Los Angeles, CA', '323-660-8999', 'M', 31000, '333000333', 3);

insert into employee
values ('123456779', 'Zin', 'B', 'Kong', '10-nov-1977', '345 Pacific Coast HWY, Long Beach, CA', '562-591-1023', 'M', 30000, '444000444', 4);

insert into employee
values ('123456778', 'Cindy', 'L', 'Lung', '11-sep-1976', '2345 Hollywood Blvd., Los Angeles, CA', '323-660-8999', 'F', 33000, '111000111', 1);

insert into employee
values ('123456777', 'Katrina', 'B', 'Sue', '12-mar-1975', '126 N.Western Ave., Los Angeles, CA', '323-660-8900', 'F', 32000, '444000444', 4);

insert into employee
values ('111000111', 'Suna', 'S', 'Mar', '1-oct-1945', '328 N.Western Ave, Los Angeles, CA', '323-660-3119', 'F', 65000, '111222333', 1);

insert into employee
values ('222000222', 'Jammy', 'A', 'Bone', '2-jan-1952', '5272 Fullerton Rd, Rowland Height, CA', '323-660-8227', 'M', 70000, '987654321', 2);

insert into employee
values ('333000333', 'Johny', 'M', 'Lui', '2-dec-1945', '503 N.Western Ave, Los Angeles, CA', '323-660-1229', 'M', 65000, '555666777', 3);

insert into employee
values ('444000444', 'Sammy', 'I', 'Hong', '1-jun-1951', '511 Fullerton Rd, Rowland Height, CA', '323-660-2222', 'M', 60000, '777888999', 4);

insert into employee
values ('111222333', 'Sunita', 'S', 'Mar', '1-oct-1945', '306 N.Western Ave, Los Angeles, CA', '323-660-3999', 'F', 80000, '987654321', 1);

insert into employee
values ('333444555', 'James', 'A', 'Bone', '2-jan-1952', '5222 Fullerton Rd, Rowland Height, CA', '323-660-8567', 'M', 80000, '987654321', 2);

insert into employee
values ('555666777', 'John', 'M', 'Lui', '2-dec-1945', '523 N.Western Ave, Los Angeles, CA', '323-660-1299', 'M', 80000, '987654321', 3);

insert into employee
values ('777888999', 'Sony', 'I', 'Hong', '1-jun-1951', '5111 Fullerton Rd, Rowland Height, CA', '323-660-2222', 'M', 80000, '987654321', 4);

insert into employee
values ('987654321', 'Sumuscha', NULL, 'Teesri', '5-apr-1950', '1126 Hollywood Blvd., Los Angeles, CA', '323-660-8939', 'M', 100000, NULL, NULL);

insert into employee
```sql
WORKS_ON:

```insert into works_on
values ('123456789', 1, 29);
insert into works_on
values ('123456787', 2, 7);
insert into works_on
values ('123456786', 3, 10);
insert into works_on
values ('123456785', 4, 30);
insert into works_on
values ('123456787', 12, 7);
insert into works_on
values ('123456786', 13, 10);
insert into works_on
values ('123456785', 14, 30);
insert into works_on
values ('123456784', 15, 13);
insert into works_on
values ('123456789', 13, 12);
insert into works_on
values ('123456789', 14, 11);
insert into works_on
values ('123456789', 15, 10);
insert into works_on
values ('123456789', 16, 9);
insert into works_on
values ('123456789', 17, 19);
insert into works_on
values ('111222333', 1, 30);
insert into works_on
values ('111222333', 2, 30);
insert into works_on
values ('111222333', 3, 10);
insert into works_on
values ('111222333', 4, 20);
insert into works_on
values ('111222333', 5, 15);
insert into works_on
values ('111222333', 6, 30);```
insert into works_on
values ('111222333', 7, 10);
insert into works_on
values ('111222333', 8, 30);
insert into works_on
values ('123456780', 16, 30);
insert into works_on
values ('123456780', 17, 30);
insert into works_on
values ('123456777', 18, 30);
insert into works_on
values ('333444555', 13, 30);
insert into works_on
values ('555666777', 17, 30);
insert into works_on
values ('777888999', 11, 30);
insert into project
values (1, 'ProductY', 1);
insert into project
values (2, 'ProductZ', 1);
insert into project
values (3, 'Computerization', 2);
insert into project
values (4, 'ReorganizationP2', 3);
insert into project
values (5, 'Newbenefits', 2);
insert into project
values (6, 'SoilY', 4);
insert into project
values (7, 'SoilZ', 4);
insert into project
values (8, 'ReorganizationP1', 3);
insert into project
values (9, 'ReorganizationP3', 3);
insert into project
values (10, 'AirX', 4);
insert into project
values (11, 'SoilX', 4);
insert into project
values (12, 'ProductN', 1);
insert into project
values (13, 'ProductT', 1);
insert into project
values (14, 'WaterX', 4);
insert into project
values (15, 'ProductP', 1);
insert into project
values (16, 'ReorganizationP4', 3);
insert into project
values (17, 'WaterY', 4);
insert into project
values (18, 'ProductR', 1);
insert into project
values (19, 'WaterZ', 4);
insert into project
values (20, 'ReorganizationP5', 3);

LOCATION:
insert into location(dno, dlocation)
values (1, 'Houston');
insert into location(dno, dlocation)
values (2, 'Houston');
insert into location(dno, dlocation)
values (3, 'Houston');
insert into location(dno, dlocation)
values (4, 'Houston');
insert into location(dno, dlocation)
values (1, 'Stafford');
insert into location(dno, dlocation)
values (2, 'Stafford');
insert into location(dno, dlocation)
values (3, 'Stafford');
insert into location(dno, dlocation)
values (4, 'Stafford');
insert into location(dno, dlocation)
values (1, 'Bellari');
insert into location(dno, dlocation)
values (1, 'Sugarland');
insert into location(dno, dlocation)
values (1, 'Soilland');

insert into location(dno, dlocation)
values (1, 'Fullerton');

insert into location(dno, dlocation)
values (1, 'Hollywood');
APPENDIX C: SOURCE CODE OF SQL SERVER 7.0 AND ORACLE 8

The COMPANY1 Database Source Code in SQL SERVER 7.0

ERROR MESSAGES:
/*Error messages used to implement triggers in both COMPANY1 and COMPANY2 databases.*/
sp_addmessage 50100, 15, 'Employee number cannot be null.'
go
sp_addmessage 50101, 15, 'Employee first name cannot be null.'
go
sp_addmessage 50102, 15, 'Employee last name cannot be null.'
go
sp_addmessage 50103, 15, 'Employee birthdate cannot be null.'
go
sp_addmessage 50104, 15, 'Employee address cannot be null.'
go
sp_addmessage 50105, 15, 'The salary of employee cannot be null.'
go
sp_addmessage 50106, 15, 'Employee number must be unique.'
go
sp_addmessage 50107, 15, 'The gender of employee must be either M male or F female.'
go
sp_addmessage 50108, 15, 'Salary cannot be less than zero.'
go
sp_addmessage 50109, 15, 'Supervisor must be an employee in EMPLOYEE table or null value.'
go
sp_addmessage 50110, 15, 'Department number must be a department in DEPARTMENT table or null value.'
go
sp_addmessage 50111, 15, 'A default department (department number =10) cannot be deleted.'
go
sp_addmessage 50112, 15, 'An employee cannot have a higher salary than the manager of department.'
go
sp_addmessage 50113, 15, 'The salary of employee cannot be higher than that of his/her supervisor salary.'
go
sp_addmessage 50114, 15, 'The salary of employee, who is not a manager of department, cannot increase more than 15%.'
go
sp_addmessage 50115, 15, 'Employee who is more than 60 years old cannot get a raise in salary.'
go
sp_addmessage 50116, 15, 'The age of employee must be between 18 and 65 years old.'
go
sp_addmessage 50117, 15, 'Department number cannot be null.'
go
sp_addmessage 50200, 15, 'Department number cannot be null.'
go
sp_addmessage 50201, 15, 'Department number must be unique.'
go
sp_addmessage 50202, 15, 'The manager of department must be an
employee in EMPLOYEE table.

sp_addmessage 50203, 15, 'Department number or department location cannot be null.'

sp_addmessage 50204, 15, 'The name of department must be unique.'

sp_addmessage 50205, 15, 'The name of department cannot be null.'

sp_addmessage 50206, 15, 'Department number must be an existing department.'

sp_addmessage 50250, 15, 'Department location cannot be null.'

sp_addmessage 50251, 15, 'Department and location must be unique.'

sp_addmessage 50300, 15, 'Employee number must be an employee in EMPLOYEE table.'

sp_addmessage 50301, 15, 'Project number must be a project in PROJECT table.'

sp_addmessage 50302, 15, 'Employee and Project number must be unique.'

sp_addmessage 50303, 15, 'Employee, not manager, cannot work more than 10 projects.'

sp_addmessage 50304, 15, 'Employee number or project number cannot be null.'

sp_addmessage 50305, 15, 'Employee work hours cannot be less than zero.'

sp_addmessage 50401, 15, 'Project number cannot be null.'

sp_addmessage 50402, 15, 'Project number must be unique.'

sp_addmessage 50403, 15, 'The name of project must be unique.'

sp_addmessage 50404, 15, 'A project must be controlled by a department.'

sp_addmessage 50405, 15, 'A department must be an existing department.'

sp_addmessage 50406, 15, 'The name of project cannot be null.'
TRIGGERS:

DEPARTMENT:

/* Trigger for a delete operation on the DEPARTMENT table. */
CREATE TRIGGER [del_dept_trgl] ON DEPARTMENT
FOR DELETE
AS

declare @dno int,
    @default_dno int,
    @num_del int,
    @dname varchar(40),
    @empno varchar(11),
    @pnumber int,
    @pname varchar(30)

/* Determine how many rows are deleted. */
select @num_del = @@rowcount
if @num_del = 0
    return

/* Cursor keeps the information of the deleted department. */
declare cur_del_department cursor
for select d.dno, d.dname
from deleted d
open cur_del_department
fetch next from cur_del_department into @dno, @dname
while (@@fetch_status <> -1 )
begin
    /* Delete a default department (dno = 10) is not allowed. */
    if @dno = 10
        begin
            raiserror( 50112 ,15, 1)
            rollback tran
            return
        end

    /* If a deleted department controls projects, set the department
    number in project records to a default department (dno = 10). */
    if exists
        (select * from project where dno = @dno)
        begin
            declare cur_default_proj cursor
            for select pnumber, pname
            from project
            where dno = @dno
            open cur_default_proj
            fetch next from cur_default_proj into @pnumber,
            @pname
            while (@@fetch_status <> -1 )
                begin
                    insert into dept_of_proj_aud
                    values(@pnumber, @pname, @dno, @dname,
            end
        end

end
getdate());
fetch next from cur_default_proj into @pnumber,
@pname
end /*while*/
deallocate cur_default_proj
select @default_dno = 10
update project
set dno = @default_dno
where dno = @dno
end

/* If a deleted department has some employees working on it, set the department number in employee records to null. */
if exists
(select * from employee where dno = @dno)
begin
/* Cursor keeps the information of employees who works on to the deleted department. */
declare cur_emp_on_dept cursor
for select empno
from employee
where dno = @dno
open cur_emp_on_dept
fetch next from cur_emp_on_dept into @empno
while (@@fetch_status <> -1 )
begin
/* Keep the information of supervisees whose supervisor is deleted in to the supervisor_aud audit table. */
insert into department_aud
values(@dno, @dname, @empno, getdate())
fetch next from cur_emp_on_dept into @empno
end /*while*/
deallocate cur_emp_on_dept
update employee
set dno = NULL
where dno = @dno
end

/* If a department is deleted from the DEPARTMENT table, all locations of this department are also deleted. */
if exists
(select * from location where dno = @dno)
begin
delete location
where dno = @dno
end
fetch next from cur_del_department into @dno, @dname
deallocate cur_del_department
/*Trigger for an insert operation on the DEPARTMENT table.*/
CREATE TRIGGER [ins_dept_trgl] ON DEPARTMENT
FOR INSERT
AS
  declare  @dno int,
          @counttt int,
          @counttl int
  select @dno = i.dno
          from inserted i

/* Check a primary key constraint on the department number field. */
/* Display an error message if the department number is null. */
if @dno is NULL
begin
  raiserror(50200, 15, 1)
  rollback tran
  return
end
else
begin
  /* Display an error message if the department number is not unique. */
  select @counttt = count(*)
          from department
          where dno = @dno

  if @counttt > 1
  begin
    raiserror(50201, 15, 1)
    rollback tran
    return
  end
/*end if*/
/*end else */
/* Check not null and unique key constraints on the department name. */
declare @dname varchar(40)
select @dname = i.dname
from inserted i

if @dname is NULL
begin
  raiserror(50205, 15,1)
  rollback tran
  return
end
else
begin
  select @counttl = count(*)
          from department
          where upper(dname) = upper(@dname)

  if @counttl > 1
  begin

  end
/*end if*/
/*end else*/
raiserror(50204, 15, 1)
rollback tran
return
end

/* Check a foreign key constraint: the manager of department
must be an employee in the EMPLOYEE table. */
/* Display an error message if an inserted department's manager
is not an existing employee. */
declare @mgrssn varchar(11)
select @mgrssn = i.mgrssn
from inserted i
if (@mgrssn is not NULL) and exists (select 'true'
from inserted i
left join employee e
on e.empno = i.mgrssn
where e.empno is NULL )
begin
raiserror(50202, 15, 1)
rollback tran
return
end
/* Trigger for an update operation on the DEPARTMENT table. */
CREATE TRIGGER [upd_dept_trgl] ON DEPARTMENT
FOR UPDATE
AS
declare @dno int, @old_dno int,
@countt int, @num_updated int
/* Determine how many rows were updated. */
select @num_updated = @@rowcount
if @num_updated = 0
    return
/* Update the department number. */
if update(dno)
    begin
        /* Check a primary key constraint: a department number
cannot be updated to a null value. */
        select @dno = i.dno
            from inserted i
        if @dno is NULL
            begin
                raiserror(50200, 15, 1)
                rollback tran
                return
            end
        /* Check a primary key constraint: a department number
cannot be updated to an existing one. */
        select @countt = count(*)
            from department
            where dno = @dno
        if @countt > 1
            begin
                raiserror(50201, 15, 1)
                rollback tran
                return
            end
        /* If this department have any employee working on it,
update the department number in employee records to the new one. */
        select @old_dno = d.dno
            from deleted d
        if exists ( select * from employee where dno = @old_dno)
            begin
                update employee
                set dno = i.dno
                from employee e, inserted i, deleted d
                where d.dno = e.dno
            end
        /* If this department controls projects, update the
department number in project records to the new one. */
        if exists (select * from project where dno = @old_dno)
            begin
                update project
                set dno = i.dno
                from project p, inserted i, deleted d
                where d.dno = p.dno
            end
        /* Update the department in the department locations. */
update location
set dno = i.dno
from location l, inserted i, deleted d
where d.dno = l.dno

end /* Update department number. */

/* Update department name/
if update(dname)
begin
/* Check a unique key constraint on the department name:
If the name of department is updated to an existing one. */
declare @dname varchar(40)
select @dname = i.dname
from inserted i
if @dname is null
begin
raiserror(50205, 15, 1)
rollback tran
return
end
else
begin
select @countt = count(*)
from department
where upper(@dname) = upper(dname)
if @countt > 1
begin
raiserror(50204, 15, 1)
rollback tran
return
end
end /*Update department name. */

/*Update the mgrssn. */
if update(mgrssn)
begin
declare @mgrssn int
select @mgrssn = i.mgrssn
from inserted i
/* Check a foreign key constraint: the manager of
department must be updated to an existing employee. */
if (@mgrssn is not NULL) and exists
( select 'true'
from inserted i
left join employee e
on e.empno = i.mgrssn
where e.empno is NULL )
begin
raiserror(50202, 15, 1)
rollback tran
return
end
end /* Update the mgrssn. */
EMPLOYEE:

/*Trigger for a delete operation on the EMPLOYEE table.*/
CREATE TRIGGER [del_emp_trgl] ON [EMPLOYEE]
FOR DELETE
AS

declare @empno varchar(11),
@num_del int,
@supervisee varchar(11),
@dno int,
@dname varchar(40)

/*Determine how many rows are deleted. */
select @num_del = @@rowcount
if @num_del = 0
return

/*Cursor keeps the information of the deleted employees.*/
declare cur_del_emp cursor
for select d.empno
from deleted d
open cur_del_emp
fetch next from cur_del_emp into @empno

while (@@fetch_status <> -1 )
begin
    /* If a supervisor is deleted from the EMPLOYEE table, the
    supervisor ssn of the supervisees will be set to null. */
    if exists (select 'true'
    from deleted d
    join employee e
    on d.empno = e.ssn)
    begin
        /* Cursor keeps the information of supervisees whose
        supervisor is deleted from the table.*/
        declare cur_supervisee cursor
        for select empno
        from employee
        where ssn = @empno

        open cur_supervisee
        fetch next from cur_supervisee into @supervisee
        while (@@fetch_status <> -1 )
        begin
         /* Keep the information of supervisees whose
        supervisor is deleted in to the supervisor_aud audit table. */
            insert into supervisor_aud
            values(@empno, @supervisee, getdate())
        end /*while*/
        deallocate cur_supervisee

    end /*if*/

    /* Update the supervisor ssn to null. */
    update employee...
set ssn = null
where ssn = @empno
end

/* If an employee, a manager of department, is deleted from the EMPLOYEE table, the department's manager mgrssn in the DEPARTMENT table will be set to null. */
if exists
     (select *
      from department dept
      where dept.mgrssn = @empno)
begin
/* Cursor keeps the information of the department when the manager of department is deleted. */
declare cur_manager cursor
for select dno, dname
from department
where mgrssn = @empno
open cur_manager
fetch next from cur_manager into @dno, @dname
while (@@fetch_status <> -1)
begin
    /*keep the information of the department's manager in the manager_aud audit table when a deleted employee is a manager. */
    insert into manager_aud
    values(@empno, @dno, @dname, getdate())
    fetch next from cur_manager into @dno, @dname
end /*while*/
deallocate cur_manager

/* Update the manager of department to null. */
update department
set mgrssn = null
where mgrssn = @empno
end

/* When an employee is deleted from the EMPLOYEE table, all employee work records in the WORKS_ON tables are also deleted. */
if exists
     (select * from works_on where empno = @empno)
begin
    delete from works_on
    where empno = @empno
end
fetch next from cur_del_emp into @empno
end /*end while*/
deallocate cur_del_emp
/*Trigger for an insert operation on the EMPLOYEE table. */
CREATE TRIGGER [ins_emp_trgl] ON [EMPLOYEE]
FOR INSERT
AS
    declare  @empno varchar(11),
            @countt int
    select @empno = i.empno
        from inserted i

    /* Check a primary key constraint on the employee number field.*/
    /* Display an error message if an employee number is null. */
    if @empno is NULL
        begin
            raiserror(50100, 15, 1)
            rollback tran
            return
        end
    /* Display an error message if an employee number is not unique. */
    else
        begin
            select @countt = count(*)
                from employee
                    where empno= @empno
            if @countt > 1
                begin
                    raiserror(50106, 15, 1)
                    rollback tran
                    return
                end
        end

    /* Check a not null constraint on the employee's first name. */
    /* Display an error message if the first name of employee is null. */
    declare @fname varchar(11)
    select @fname = i.fname
        from inserted i
    if @fname is NULL
        begin
            raiserror(50101, 15, 1)
            rollback tran
            return
        end

    /* Check a not null constraint on the employee's last name. */
    /* Display an error message if the last name of employee is null. */
    declare @lname varchar(11)
    select @lname = i.lname
        from inserted i
    if @lname is NULL
        begin

raiserror(50102, 15, 1)
rollback tran
return
end

/* Check a not null constraint on the employee's birth date. */
/* Display an error message if the birth date of employee is
null. */
declare @bdate varchar(11)
select @bdate = i.bdate
from inserted i
if @bdate is NULL
begin
raiserror(50103, 15, 1)
rollback tran
return
end

/* Check a user-defined constraint: the age of employee
must be between 18 and 65. */
declare @age int
select @age = datediff(year, i.bdate, getdate())
from inserted i
if (@age <18) or (@age > 65)
begin
raiserror(50119, 15, 1)
rollback tran
return
end

/* Check a not null constraint on the employee's address. */
/* Display an error message if the address of employee is null. */
declare @address varchar(11)
select @address = i.address
from inserted i
if @address is NULL
begin
raiserror(50104, 15, 1)
rollback tran
return
end

/* Check a check constraint: sex must be either 'M' male or 'F'
female. */
declare @sex char(1)
select @sex = i.sex
from inserted i
if (@sex is not NULL)
begin
if ((upper(@sex) != 'F') and (upper(@sex) != 'M'))
begin
raiserror(50107, 15, 1)
rollback tran
return
end

/* Check a foreign key constraint: the supervisor ssn must be an existing employee. */
/* Display an error message if the inserted supervisor is not an employee in the EMPLOYEE table. */
declare @ssn varchar(11)
select @ssn = i.ssn
from inserted i
if (@ssn is not NULL) and exists ( select 'true'
    from inserted i
    left join employee e
    on e.empno = i.ssn
    where e.empno is NULL )
begin
    raiserror(50110, 15, 1)
    rollback tran
    return
end

/* Check a foreign key constraint: the department in the EMPLOYEE table must exist in the DEPARTMENT table. */
/* Display an error message if the inserted department is not an existing department. */
declare @dno int
select @dno = i.dno
from inserted i
if (@dno is not NULL) and exists ( select 'true'
    from inserted i
    left join department dept
    on dept.dno = i.dno
    where dept.dno is NULL )
begin
    raiserror(50111, 15, 1)
    rollback tran
    return
end

/* Check a not null constraint on the employee's salary. */
/* Display an error message if the salary of employee is null. */
declare @newsal money
select @newsal = i.salary
from inserted i
if @newsal is NULL
begin
    raiserror(50105, 15, 1)
    rollback tran
    return
end
/* Check a check constraint: the salary of employee cannot be less than zero. */
if @newsal < 0
    begin
        raiserror(50108, 15, 1)
        rollback tran
        return
    end

/* Check a user-defined constraint: the supervisee cannot have a higher salary than his/her supervisor. */
declare @ssnsal money,
    @mgrssnsal money
select @ssnsal = s.salary
from inserted i, employee s
where i.ssn = s.empno
if @newsal > @ssnsal
    begin
        raiserror(50116, 15, 1)
        rollback tran
        return
    end

/* Check a user-defined constraint: an employee cannot have a higher salary a manager of department. */
else
    begin
        select @mgrssnsal = salary
            from employee
            where empno in (select mgrssn
                from inserted i, department d
                where i.dno = d.dno);

        if @newsal > @mgrssnsal
            begin
                raiserror(50115, 15, 1)
                rollback tran
                return
            end
    end
/* Trigger for an update operation on the EMPLOYEE table. */

CREATE TRIGGER [upd_emp_trgl] ON [EMPLOYEE]
FOR UPDATE
AS

declare @empno varchar(11),
        @old_empno varchar(11),
        @countt int,
        @num_updated int

/* Determine how many rows are updated. */
select @num_updated = @@rowcount
if @num_updated = 0
    return

/* Update the employee number */
if update(empno)
begin
    /* Check a primary key constraint: an employee number
    cannot be updated to a null value. */
    select @empno = i.empno
    from inserted i
    if @empno is NULL
    begin
        raiserror(50100, 15, 1)
        rollback tran
        return
    end

    /* Check a primary key constraint: an employee number
    cannot be updated to an existing one. */
    select @countt = count(*)
    from employee
    where empno = @empno
    if @countt > 1
    begin
        raiserror(50106, 15, 1)
        rollback tran
        return
    end

    /* If this employee is a supervisor ssn, then update the
    supervisor ssn to a new value. */
    select @old_empno = d.empno
    from deleted d
    if exists (select * from employee where ssn = @old_empno)
    begin
        update employee
        set ssn = i.empno
        from employee e, inserted i, deleted d
        where d.empno = e.ssn
    end

    /* If this employee is a manager mgrssn of department,
    then update the manager of department to a new value. */
if exists (select * from department where mgrssn = @old_empno)
begin
    update department
    set mgrssn = i.empno
    from department dept, inserted i, deleted d
    where dept.mgrssn = d.empno
end

/* If this employee works on projects, then update employee number in the WORKS_ON table. */
if exists (select * from works_on where empno = @old_empno)
begin
    update works_on
    set empno = i.empno
    from works_on w, inserted i, deleted d
    where d.empno = w.empno
end

end /* Update employee number */

/* Update the first name of employee. */
if update(fname)
begin
    /* Check a not null constraint: display an error message if the first name is updated to a null value. */
    declare @fname varchar(20)
    select @fname = i.fname
    from inserted i
    if @fname is NULL
    begin
        raiserror(50101, 15, 1)
        rollback tran
        return
    end
end /* Update the first name of employee. */

/* Update the last name of employee. */
if update(lname)
begin
    /* Check a not null constraint: display an error message if the last name is updated to a null value. */
    declare @lname varchar(20)
    select @lname = i.lname
    from inserted i
    if @lname is NULL
    begin
        raiserror(50102, 15, 1)
        rollback tran
        return
    end
end /* Update the last name of employee. */

/* Update the birthdate of employee. */
if update(bdate)
begin
/* Check a not null constraint: display an error message if the birth date is updated to a null value. */
declare @empbdate varchar(20)
select @empbdate = i.bdate
from inserted i
if @empbdate is NULL
begin
    raiserror(50103, 15, 1)
    rollback tran
    return
end

/* Check a user-defined constraint: the age of employee must be between 18 and 65. */
declare @age int
select @age = datediff(year, i. bdate, getdate())
from inserted i
if (@age <18) or (@age >65)
begin
    raiserror(50119, 15, 1)
    rollback tran
    return
end

end /* Update the birth date of employee. */

/* Update the address of employee. */
if update(address)
begin
    /* Check a not null constraint: display an error message if the address is updated to a null value. */
declare @address varchar(20)
select @address = i.address
from inserted i
if @address is NULL
begin
    raiserror(50104, 15, 1)
    rollback tran
    return
end
end /* Update the address of employee. */

/*Update the sex attribute. */
if update(sex)
begin
    /* Check the gender of employee: display an error message if the sex of employee is not updated to 'F' or 'M'. */
declare @sex char(1)
select @sex = i.sex
from inserted i
if (@sex is not NULL)
begin
    if ((upper(@sex) != 'F') and (upper(@sex) != 'M'))
    begin
        raiserror(50107, 15, 1)
        rollback tran
        return
    end

end 139
end /* Updating sex */

/* Update the supervisor ssn. */
if update(ssn)
begin
/* Check a foreign key constraint: the supervisor
ssn must be updated to an existing employee. */
declare @ssn varchar(11)
select @ssn = i.ssn
from inserted i
if (@ssn is not NULL) and exists ( select 'true'
from inserted i
left join employee e
on e.empno = i.ssn
where e.empno is NULL )
begin
raiserror(50110, 15, 1)
rollback tran
return
end
end /* Update the ssn */

/* Update the department number dno. */
if update(dno)
begin
/* Check a foreign key constraint: the department must be
updated to an existing department in the DEPARTMENT table. */
declare @dno int
select @dno = i.dno
from inserted i
if (@dno is not NULL) and exists ( select 'true'
from inserted i
left join department dept
on i.dno = dept.dno
where dept.dno is NULL )
begin
raiserror(50111, 15, 1)
rollback tran
return
end
end /* Update the dno. */

/* Update the salary. */
if update(salary)
begin
declare @newsal money,
@oldsal money,
@ssnsal money, /* the supervisor's salary. */
@mrsalsal money, /* the manager's salary. */
@bdate datetime,
@sal money

/* Check a not null constraint: the employee's salary
cannot be updated to a null value. */
select @sal = i.salary
from inserted i

if @sal is NULL
begin
  raiserror(50105, 15, 1)
  rollback tran
  return
end

/* Check a check constraint: the salary of employee cannot be less than zero. */
if @sal < 0
begin
  raiserror(50108, 15, 1)
  rollback tran
  return
end

/*Cursor keeps the information of updating the salary of employees. */
declare cur_upd_sal cursor
for select i.empno, i.salary, d.salary, i.bdate
from inserted i inner join deleted d
on i.empno = d.empno
open cur_upd_sal
fetch next from cur_upd_sal into ©empno, ©newsal, ©oldsal,
@bdate

while (@@fetch_status <> -1 )
begin
  /* Check a user-defined constraint: the employee, who is more than 60 years old, cannot get a raise in salary. */
  if datediff(year, @bdate, getdate()) > 60
  begin
    raiserror(50118, 15, 1)
    rollback tran
    return
  end

  /* Check a user-defined constraint: the salary of employee, not manager, cannot be increased by more than 15%. */
  if (@newsal > (@oldsal * 1.15 )) and
    not exists (select 'true'
                from employee
                where @empno in
                (select distinct(mgrssn)
                 from department) )
  begin
    raiserror(50117, 15, 1)
    rollback tran
    deallocate cur_upd_sal
    return
  end
else
  begin
    /* Check a user-defined constraint: a supervisee cannot have a higher salary than his/her supervisor. */

select @ssnsal = s.salary
from inserted i, employee s
where i.ssn = s.empno

if @newsal > @ssnsal
begin
    raiserror(50116, 15, 1)
    rollback tran
    return
end
else
begin
    /* Check a user-defined constraint: an employee cannot have a higher salary than the manager of department. */
    select @mgrssnsal
    = salary
    from employee
    where empno in (select mgrssn
    from inserted i, department d
    where i.dno = d.dno);

    if @newsal > @mgrssnsal
    begin
        raiserror(50115, 15, 1)
        rollback tran
        return
    end
end

/* Keep an update of salary in the emp_sal_aud audit table. */
insert into emp_sal_aud
values(@empno, @oldsal, @newsal, getdate());

fetch next from cur_upd_sal into @empno, @newsal, @oldsal, @bdate
end /*while*/
deallocate cur_upd_sal
end /*update the salary */
CREATE TRIGGER \[ins_loc_trgl\] ON [LOCATION]
FOR INSERT
AS

declare @dno int,
        @dloc varchar(30),
        @countt int

select @dno = i.dno
    from inserted i

select @dloc = i.dlocation
    from inserted i

/* Check a not null constraint on the primary key values: the
department number and department location cannot be NULL. */
if (@dno is NULL) or (@dloc is NULL)
begin
    raiserror(50203, 15, 1)
    rollback tran
    return
end

/* Check a foreign key constraint: a department must exist in
the DEPARTMENT table. */
/* Display an error message if the department number is not an
existing department. */
if (@dno is not NULL) and exists ( select 'true'
    from inserted i
        left join department dept
            on dept.dno = i.dno
        where dept.dno is NULL )
begin
    raiserror(50206, 15, 1)
    rollback tran
    return
end

/* Check a primary key constraint on both department number and
department location. */
select @countt = count(*)
    from location l
    where @dno = l.dno and @dloc = l.dlocation
if @countt > 1
begin
    raiserror(50251, 15, 1)
    rollback tran
    return
end
/* Trigger for an update operation on the LOCATION table. */
CREATE TRIGGER [upd_loc_trgl] ON [LOCATION]
FOR UPDATE
AS
declare @dno int, @dloc varchar (30),
        @num_updated int, @countt int,
        @counttl int, @countt2 int

select @dloc = i.dlocation
        from inserted i
select @dno = i.dno
        from inserted i

/* Determine how many rows are updated. */
select @num_updated = @@rowcount
if @num_updated = 0
    return

/* Update the department number. */
if update(dno)
begin
    declare @count int
    select @count = count(*) from inserted
    /* Check a not null constraint on the primary key value. */
    /* Display an error message if the department number is updated
to a null value. */
    select @dno = i.dno
    from inserted i
    if @dno is NULL
        begin
            raiserror(50200, 15, 1)
            rollback tran
            return
        end
    /* Check a foreign key constraint: a department must exist
in the DEPARTMENT table. */
    /* Display an error message if the department is not
updated to an existing one. */
    if (@dno is not NULL) and exists ( select 'true'
        from inserted i
        left join department dept
        on dept.dno = i.dno
        where dept.dno is NULL )
        begin
            raiserror(50206, 15, 1)
            rollback tran
            return
        end
    /* Check a primary key constraint on both department
number and department location when update a department number.*/
    select @counttl = count(distinct(1.dlocation))
    from location 1 inner join inserted i
    on 1.dno = i.dno

    select @countt2 = count(1.dlocation)
    from location 1 inner join inserted i
    on 1.dno = i.dno
if (@count * @countt1) <> @countt2
begin
raiserror(50251, 15, 1)
rollback tran
return
end
end /* Update department number. */

/* Update the department location. */
if update(dlocation)
begin
/* Check a not null constraint on the primary key value. */
/* Display an error message if the department location is
updated to a null value. */
select @dloc = i.dlocation
from inserted i
if @dloc is NULL
begin
raiserror(50250, 15, 1)
rollback tran
return
end
/* Check a primary key constraint on both department and
location when update a department location. */
select @countt1 = count(distinct(l.dno))
from location 1 inner join inserted i
on l.dlocation = i.dlocation

select @countt2 = count(l.dlocation)
from location 1 inner join inserted i
on l.dlocation = i.dlocation

if @countt1 <> @countt2
begin
raiserror(50251, 15, 1)
rollback tran
return
end
end /* Updating department location */

/* Check a primary key constraint when updating both department
number and department location. */
if update(dno) or update(dlocation)
begin
select @countt = count(*)
from location 1
where @dno = l.dno
and @dloc = l.dlocation

if @countt > 1
begin
raiserror(50251, 15, 1)
rollback tran
return
end
end
PROJECT:

/* Trigger for a delete operation on the PROJECT table.*/
CREATE TRIGGER [del_proj_trgl] ON [PROJECT]
FOR DELETE
AS

declare @pnum int,
        @pname varchar(30),
        @dno int,
        @dname varchar(40),
        @mgrssn varchar(11),
        @num_del int

/* Determine how many rows are deleted. */
select @num_del = @@rowcount
if @num_del = 0
  return

/* Cursor keeps the information of the deleted projects. */
declare cur_del_proj cursor
for select d.pnumber, d.pname, d.dno
from deleted d
open cur_del_proj
fetch next from cur_del_proj into @pnum, @pname, @dno
while (@@fetch_status <> -1 )
begin
  /* If the deleted project has employees working on it, */
delete work records of this project. */
  if exists
    (select * from works_on where pnumber = @pnum)
  begin
    delete from works_on
    where pnumber = @pnum
  end

  /* Keep the deleted project information in the project_aud */
audit table. */
  select @dname = dname from department where dno = @dno
  select @mgrssn = mgrssn from department where dno = @dno
  insert into project_aud
  values(@pnum, @pname, @dno, @dname, @mgrssn, getdate());

  fetch next from cur_del_proj into @pnum, @pname, @dno
end
/*while*/
deallocate cur_del_proj
/* Trigger for an insert operation on the PROJECT table.*/
CREATE TRIGGER [ins_proj_trgl] ON [PROJECT]
FOR INSERT
AS
    declare @pnum int,
    @pname varchar(30),
    @countt int
    select @pnum = i.pnumber
    from inserted i

    /* Check a primary key constraint on the project number field. */
    /* Display an error message if a project number is null. */
    if @pnum is NULL
    begin
        raiserror(50401, 15, 1)
        rollback tran
        return
    end
    else
    begin
        /* Display an error message if the project number is not unique. */
        select @countt = count(*)
        from project
        where pnumber = @pnum
        if @countt > 1
        begin
            raiserror(50402, 15, 1)
            rollback tran
            return
        end
    end

    /* Check not null and unique key constraints on the project name. */
    /* Display an error message if the project name is null. */
    select @pname = i.pname
    from inserted i
    if @pname is null
    begin
        raiserror(50406, 15,1)
        rollback tran
        return
    end

    /*Display an error message if the name of project is duplicated.*/
    select @countt = count(*)
    from project
    where upper(pname) = upper(@pname)
    if @countt > 1
begin
    raiserror(50403, 15, 1)
    rollback tran
    return
end

/* Check constraints on the department number field. */
declare @dno int
select @dno = i.dno
    from inserted i

/* Check a not null constraint: a project must be controlled by a department. It cannot be null. */
if @dno is null
    begin
        raiserror(50404, 15, 1)
        rollback tran
        return
    end
else
    begin
        /* Check a foreign key constraint: a department must exist in the DEPARTMENT table. */
        /* Display an error message if an inserted department does not exist in the DEPARTMENT table. */
        if (@dno is not NULL) and exists ( select 'true'
            from inserted i
            left join department dept
            on i.dno = dept.dno
            where dept.dno is NULL )
            begin
                raiserror(50405, 15, 1)
                rollback tran
            end
    end
/*Trigger for an update operation on the PROJECT table.*/
CREATE TRIGGER [upd_proj_trgl] ON [PROJECT]
FOR UPDATE
AS

declare @pnum int, @old_pnum int,
        @countt int, @num_updated int

/*Determine how many rows are updated.*/
select @num_updated = @@rowcount
if @num_updated = 0
   return

/* Update the project number */
if update(pnumber)
begin
   /* Check a primary key constraint on the project number field. */
   /* Display an error message if this project number is updated to a null value. */
   select @pnum = i.pnumber
      from inserted i
   if @pnum is NULL
      begin
         raiserror(50401, 15, 1)
         rollback tran
         return
      end
   /* Display an error message if this project number is updated to an existing one. */
   select @countt = count(*)
      from project
      where pnumber = @pnum
   if @countt > 1
      begin
         raiserror(50402, 15, 1)
         rollback tran
         return
      end
   /* If this project has employees working on it, update work records in the WORKS_ON table. */
   select @old_pnum = d.pnumber
      from deleted d
   if exists (select * from works_on where pnumber = @old_pnum)
      begin
         update works_on
            set pnumber = i.pnumber
            from works_on w, inserted i, deleted d
            where d.pnumber = w.pnumber
      end
   end /* Updating project number. */

   /* Update the project name. */
   if update(pname)
      begin
         /* Check not null and unique key constraints on the project name.
/* Display an error message if the project name is updated to null. */
declare @pname varchar(30)
select @pname = i.pname
from inserted i
if @pname is null
begin
    raiserror(50406, 15, 1)
    rollback tran
    return
end

/* Display an error message if the name of project is updated to an existing project's name. */
select @countt = count(*)
from project
where upper(pname) = upper(@pname)
if @countt > 1
begin
    raiserror(50403, 15, 1)
    rollback tran
    return
end
end /* Update the project name. */

/* Update the department number. */
if update(dno)
begin
declare @dno int
select @dno = i.dno
from inserted i

/* Check a constraint: a project must be controlled by a department. */
if @dno is null
begin
    raiserror(50404, 15, 1)
    rollback tran
    return
end
else
begin
    /* Check a foreign key constraint: A department number must be updated to an existing department. */
    if (@dno is not NULL) and exists ( select 'true'
        from inserted i
        left join department dept
        on i.dno = dept.dno
        where dept.dno is NULL )
    begin
        raiserror(50405, 15, 1)
        rollback tran
        return
    end
end
end /* Update the department number. */
WORKS_ON:

/*Trigger executed when inserting data in WORKS_ON table.*/
CREATE TRIGGER [ins_works_on_trgl] ON [WORKS_ON]
FOR INSERT
AS

declare @empno varchar(11),
        @pnum int,
        @countt int

select @empno = i.empno
    from inserted i

select @pnum = i.pnumber
    from inserted i

/* The composite key, empno and pnumber, cannot be NULL. */
if (@empno is NULL) or (@pnum is NULL)
begin
    raiserror(50304, 15, 1)
    rollback tran
    return
end

/* Foreign key, the empno attribute, must be in EMPLOYEE parent table. */
if (@empno is not NULL) and exists ( select 'true'
    from inserted i
    left join employee e
    on e.empno = i.empno
    where e.empno is NULL )
begin
    raiserror(50300, 15, 1)
    rollback tran
    return
end

/* Foreign key, the pnumber attribute, must be in PROJECT parent table. */
if (@pnum is not NULL) and exists ( select 'true'
    from inserted i
    left join project p
    on i.pnumber = p.pnumber
    where p.pnumber is NULL )
begin
    raiserror(50301, 15, 1)
    rollback tran
    return
end

/* Both empno and pnumber must be unique. */
select @countt = count(*)
from works_on w
where @empno = w.empno
and @pnum = w.pnumber
if @countt > 1
begin
    raiserror(50302, 15, 1)
    rollback tran
    return
end

/* Display error message if the hour is less than zero. */
declare @hours real
select @hours = i.hours
from inserted i
if @hours < 0
begin
    raiserror(50305, 15, 1)
    rollback tran
    return
end

/* An employee, not a manager, cannot work more than 10
projects. */
select @countt = count(*)
from works_on w
where @empno = w.empno
group by w.empno
if (@countt > 10) and not exists (select 'true'
    from department dept
    where dept. mgrssn = @empno)
begin
    raiserror(50303, 15, 1)
    rollback tran
    return
end
/* Trigger for an update operation on the WORKS_ON table. */
CREATE TRIGGER [upd_works_on_trgl] ON [WORKS_ON]
FOR UPDATE
AS

declare @empno varchar(11),
         @pnum int,
         @num_updated int,
         @countt int,
         @countt1 int,
         @countt2 int

/* Determine how many rows are updated. */
select @num_updated = @rowcount
if @num_updated = 0
   return

   declare @count int
   select @count = count(*) from inserted

select @empno = i.empno
   from inserted i

select @pnum = i.pnumber
   from inserted i

/* Update the employee number. */
if update(empno)
begin
   /* Check a not null constraint on the primary key value. */
   if @empno is NULL
      begin
         raiserror(50100, 15, 1)
         rollback tran
         return
      end

   /* Check a foreign key constraint: an employee must exist
   in the EMPLOYEE table. */
   /* Display an error message if this employee is not
   updated to an existing employee. */
   if (@empno is not NULL) and exists ( select 'true'
      from inserted i
      left join employee e
      on e.empno = i.empno
      where e.empno is NULL )
      begin
         raiserror(50300, 15, 1)
         rollback tran
         return
      end

   select @countt1 = count(w.pnumber)
   from works_on w inner join inserted i
   on w.empno = i.empno

   select @countt2 = count(distinct(w.pnumber))
from works_on w inner join inserted i
on w.empno = i.empno

if @countt1 <> (@countt2 * @countt)
begin
  raiserror(50302, 15, 1)
  rollback tran
  return
end

select @countt = count(*)
from works_on w
where @empno = w.empno
  and @pnum = w.pnumber

if @countt > 1
begin
  raiserror(50302, 15, 1)
  rollback tran
  return
end
end /* Updating employee number */

/* Updating the project number */
if update(pnumber)
begin

select @pnum = i.pnumber
from inserted i

/* Display an error message if the project number is
updated to a null value. */
if @pnum is NULL
begin
  raiserror(50401, 15, 1)
  rollback tran
  return
end

/* Check a foreign key constraint: a project must exist in
the PROJECT table. */
/* Display an error message if the project number is not
updated to an existing project. */
if (@pnum is not NULL) and exists ( select 'true'
from inserted i
left join project p
on i.pnumber = p.pnumber
where p.pnumber is NULL )
begin
  raiserror(50301, 15, 1)
  rollback tran
  return
end

/* Check a primary key constraint on both project number
and employee number when project number is updated. */
select @countt1 = count(empno)
from works_on
where pnumber = @pnum

select @countt2 = count(distinct(empno))
from works_on
where pnumber = @pnum

if @countt1 <> @countt2
begin
    raiserror(50302, 15, 1)
    rollback tran
    return
end

select @empno = i.empno
    from inserted i

select @countt = count(*)
from works_on w
where @empno = w.empno
    and @pnum = w.pnumber

if @countt > 1
begin
    raiserror(50302, 15, 1)
    rollback tran
    return
end

end /* Update project number */

/* Check a primary constraint on both employee number and project number when both attributes are updated. */
    if update(empno) or update(pnumber)
begin
    select @countt = count(*)
    from works_on w
    where @empno = w.empno
        and @pnum = w.pnumber

    if @countt > 1
    begin
        raiserror(50302, 15, 1)
        rollback tran
        return
    end
end

/* Check a user-defined constraint: an employee, not a manager of department, cannot work more than 10 projects. */
    select @empno = i.empno
    from inserted i

select @countt = count(*)
from works_on w
where @empno = w.empno

if (@countt > 10 ) and not exists (select 'true'
from department dept
where dept.mgrssn = @empno
begin
raiserror(50303, 15, 1)
rollback tran
return
end

/* Update employee work hours */
if update(hours)
begin
  /* Check a constraint on hours: an employee work hours
  cannot be less than zero. */
  declare @hours real
  select @hours = i.hours
  from inserted i
  if @hours < 0
    begin
      raiserror(50305, 15, 1)
      rollback tran
      return
    end
end /*Update employee work hours. */
EMPLOYEE:

/*Trigger for an insert operation on the EMPLOYEE table.*/
CREATE TRIGGER [ins_emp_trg2] ON [EMPLOYEE]
FOR INSERT
AS
declare @empno varchar(11)
/* Check a user-defined constraint: The supervisee cannot have a higher salary than his/her supervisor. */
declare @ssnsal money,
@newsal money,
@mgrssnsal money

select @newsal = i.salary
from inserted i

select @ssnsal = s.salary
from inserted i, employee s
where i.ssn = s.empno

if @newsal > @ssnsal
begin
raiserror(50116, 15, 1)
rollback tran
end
else
begin

/* An employee cannot have a higher salary than the manager of department. */
select @mgrssnsal = salary
from employee
where empno in (select mgrssn
from inserted i, department d
where i.dno = d.dno);

if @newsal > @mgrssnsal
begin
raiserror(50115, 15, 1)
rollback tran
return
end

end

/* Check a user-defined constraint: the age of employee must be between 18 and 65. */
declare @age int
select @age = datediff(year,i. bdate, getdate())
from inserted i

if (@age <18) or (@age > 65)
begin
raiserror(50119, 15, 1)
rollback tran
return
end
/**Trigger for an update operation on the EMPLOYEE table.*/

CREATE TRIGGER [upd_emp_trg2] ON [EMPLOYEE]
FOR UPDATE
AS

declare @empno varchar(11),
@countt int,
@num_updated int,
@old_empno varchar(11)

/*Determine how many rows are updated. */
select @num_updated = @@rowcount
if @num_updated = 0
    return

/* Update the salary. */
if update(salary)
begin
    declare @newsal money,
    @oldsal money,
    @ssnsal money, /*the salary of supervisor. */
    @mgrssnsal money, /*the salary of the department's
    manager. */
    @bdate datetime

    /*Cursor keeps the information about an update on the
    salary of employees. */
    declare cur_upd_sal cursor
    for select i.empno, i.salary, d.salary, i.bdate
    from inserted i inner join deleted d
    on i.empno = d.empno

    open cur_upd_sal
    fetch next from cur_upd_sal into @empno, @newsal, @oldsal, @bdate
    while (@@fetch_status <> -1 )
    begin
        /*The age of employee who is more than 60 not get a raise
        in salary. */
        if datediff(year, @bdate, getdate()) > 60
            begin
                raiserror(50118, 15, 1)
                rollback tran
                return
            end

        /* The salary of employee, not manager, cannot be
        increased by more than 15%. */
        if (@newsal > (@oldsal * 1.15 )) and
        not exists (select 'true'
        from employee
        where @empno in (select
distinct(ssn) from employee)
        )
            begin
                raiserror(50117, 15, 1)
                rollback tran
            deallocate cur_upd_sal
return
else /* Supervisee cannot have a higher salary than
his/her supervisor. */
begin
    select @ssnsal = s.salary
    from inserted i, employee s
    where i.ssn = s.empno

    if @newsal > @ssnsal
        begin
            raiserror(50116, 15, 1)
            rollback tran
            return
        end
    else
        begin
            /* An employee cannot have a higher salary
            than the manager of department. */
            select @mgrssnsal = salary
            from employee
            where empno in (select mgrssn
                            from inserted i, department d
                            where i.dno = d.dno);

            if @newsal > @mgrssnsal
                begin
                    raiserror(50115, 15, 1)
                    rollback tran
                    return
                end
        end
    end

/* Keep an update salary of employee to the emp_sal_aud
audit table. */
insert into emp_sal_aud
values(@empno, @oldsal, @newsal, getdate());
fetch next from cur_upd_sal into @empno, @newsal, @oldsal,
@end /*while*/
deallocate cur_upd_sal
end /*update the salary */
/*Update the age of employee. */
if update(bdate)
begin /* The age of employee must be between 18 and 65. */
    declare @age int
    select @age = datediff(year, i. bdate, getdate())
    from inserted i
    if (@age <18) or (@age > 65)
        begin
            raiserror(50119, 15, 1)
            rollback tran
            return
        end
end /* Update the birth date. */
WORKS_ON:

/* Trigger for insert an employee work record into the WORKS_ON table. */
CREATE TRIGGER [ins_works_on_trg2] ON [WORKS_ON]
FOR INSERT
AS

declare   @empno varchar(11),
           @pnum int,
           @countt int

select   @empno = i.empno
         from inserted i

select   @pnum = i.pnumber
         from inserted i

/* Check an employee, not a manager, cannot work more than 10 projects. */
select   @countt = count(*)
         from works_on w
         where @empno = w.empno
         group by w.empno

if (@countt > 10) and not exists (select 'true'
                                        from department dept
                                        where dept.mgrssn = @empno)
begin
raiserror(50303, 15, 1)
rollback tran
return
end
CREATE TRIGGER [upd_works_on_trg2] ON [WORKS_ON]
FOR UPDATE
AS
    declare @empno varchar(11),
            @num_updated int,
            @countt int

    /* Determine how many rows are updated. */
    select @num_updated = @@rowcount
    if @num_updated = 0
        return

    /* An employee, not a manager of department, cannot work more
    than 10 projects. */
    select @empno = i.empno
        from inserted i

    select @countt = count(*)
        from works_on w
        where @empno = w.empno
        group by w.empno

    if (@countt > 10 ) and not exists (select 'true'
        from department dept
        where dept. mgrssn = @empno)
    begin
        raiserror(50303, 15, 1)
        rollback tran
        return
    end
The COMPANY1 Database Source Code in ORACLE 8

TRIGGERS:
DEPARTMENT:

CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_BDS"
BEFORE DELETE ON "COMPANY1"."DEPARTMENT"
begin
    del_dno_tab_pkg.clear_table;
end dept_bds;

CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_ADR"
AFTER DELETE ON "COMPANY1"."DEPARTMENT"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    dept_cons_pkg.dept_del(:old.dno, :old.dname);
    del_dno_tab_pkg.insert_value(:new.dno, :old.dno, :old.dname);
end dept_adr;

CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_ADS"
AFTER DELETE ON "COMPANY1"."DEPARTMENT"
declare
    rec del_dno_tab_pkg.rec_type;
    countt int;
begin
    del_dno_tab_pkg.set_iterator;
    while del_dno_tab_pkg.more_in_table loop
        rec := del_dno_tab_pkg.next_rec;
        -- Delete a default department is not allowed.
        if rec.old_dno = 10 then
            raise_application_error(proj_errors.default_dno,
            proj_errors.default_dno_msg);
        end if;
        -- If a deleted department controls projects, assign the
        default department to those projects.
        select count(*) into countt
        from project
        where dno = rec.old_dno;
        if countt > 0 then
            dept_cons_pkg.audit_proj(rec.old_dno, rec.old_dname);
            update project
            set dno = 10
            where dno = rec.old_dno;
        end if;
    end loop;
    exception
    when no_data_found then
        null;
end dept_ads;
CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_BIR"
BEFORE INSERT ON "COMPANY1"."DEPARTMENT"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
department_pkg.enforce_pk_dno_ins(:new.dno);
department_pkg.enforce_uk_dname_ins(:new.dname);
department_pkg.enforce_fk_mgr_ins(:new.mgrssn);
end dept_bir;

CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_BUS"
BEFORE UPDATE ON "COMPANY1"."DEPARTMENT"
begin
department_pkg.clear_table;
end dept_bus;

CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_AUR"
AFTER UPDATE OF "DNAME", "DNO", "MGRSSN" ON "COMPANY1"."DEPARTMENT"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
if updating ('dno') then
    department_pkg.insert_value(:new.dno, :old.dno, null, null);
department_pkg.check_field := 1;
end if;

if updating ('dname') then
    department_pkg.insert_value(:new.dno, null, null, :new.dname);
department_pkg.check_field := 2;
end if;

if updating ('mgrssn') then
    department_pkg.insert_value(null, null, :new.mgrssn, null);
department_pkg.check_field := 3;
end if;
end dept_aur;

CREATE OR REPLACE TRIGGER "COMPANY1"."DEPT_AUS"
AFTER UPDATE ON "COMPANY1"."DEPARTMENT"
begin
department_pkg.set_iterator;
while department_pkg.more_in_table loop
    if department_pkg.check_field = 1 then
        department_pkg.enforce_pk_dno_upd(department_pkg.next_rec);
    end if;

    if department_pkg.check_field = 3 then
        department_pkg.enforce_fk_mgr_upd(department_pkg.next_rec);
    end if;

    if department_pkg.check_field = 4 then
        department_pkg.enforce_uk_dname_upd(department_pkg.next_rec);
    end if;
end loop;
end dept_aus;
CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_BDS"
BEFORE DELETE ON "COMPANY1"."EMPLOYEE"
begin
    ssn_tab_pkg.clear_table;
end emp_bds;

CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_ADR"
AFTER DELETE ON "COMPANY1"."EMPLOYEE"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    emp_cons_pkg.emp_del(:old.empno);
    ssn_tab_pkg.insert_value(:new.empno, :old.empno);
end emp_adr;

CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_ADS"
AFTER DELETE ON "COMPANY1"."EMPLOYEE"
declare
    rec ssn_tab_pkg.rec_type;
    countt int;
begin
    ssn_tab_pkg.set_iterator;
    while ssn_tab_pkg.more_in_table loop
        rec := ssn_tab_pkg.next_rec;
        -- If a deleted employee is a supervisor, set the
        supervisor of supervisees to null.
        select count(*) into countt
        from employee
        where ssn = rec.old_ssn;
        if countt > 0 then
            emp_cons_pkg.audit_ssn(rec.old_ssn);
            update employee
            set ssn = null
            where ssn = rec.old_ssn;
        end if;
    end loop;
    exception
    when no_data_found then
        null;
end emp_ads;

CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_BIR"
BEFORE INSERT ON "COMPANY1"."EMPLOYEE"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    emp_cons_pkg.enforce_pk_empno_ins(:new.empno);
    emp_cons_pkg.check_fname(:new.fname);
emp_cons_pkg.check_lname(:new.lname);
emp_cons_pkg.check_bdate(:new.bdate);
emp_cons_pkg.check_address(:new.address);
emp_cons_pkg.check_salary(:new.empno,:new.salary,:new.ssn,:new.dno);
emp_cons_pkg.enforce_sex_ins(:new.sex);
emp_cons_pkg.enforce_fk_ssn_ins(:new.ssn);
emp_cons_pkg.enforce_fk_dno_ins(:new.dno);

end emp_bir;

CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_BUS"
BEFORE UPDATE ON "COMPANY1"."EMPLOYEE"
begin
  emp_tab_pkg.clear_table;
  ssn_tab_pkg.clear_table;
end emp_bus;

CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_AUR"
AFTER UPDATE OF "ADDRESS", "BDATE", "DNO", "EMPNO", "FNAME", "LNAME", "SALARY", "SEX", "SSN" ON "COMPANY1"."EMPLOYEE"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
  if updating ('empno') then
    emp_tab_pkg.insert_value(:new.empno,:old.empno,null,null,null,null,null,null,null,null);
    emp_cons_pkg.old_empno := :old.empno;
    emp_cons_pkg.new_empno := :new.empno;
    emp_tab_pkg.check_field := 1;
  end if;

  if updating ('fname') then
    emp_tab_pkg.insert_value(null,null,:new.fname,null,null,null,null,null,null,null,null);
    emp_tab_pkg.check_field := 2;
  end if;

  if updating ('lname') then
    emp_tab_pkg.insert_value(null,null,null,:new.lname,null,null,null,null,null,null,null,null);
    emp_tab_pkg.check_field := 3;
  end if;

  if updating ('bdate') then
    emp_tab_pkg.insert_value(null,null,null,null,:new.bdate,null,null,null,null,null,null,null);
    emp_tab_pkg.check_field := 4;
  end if;

  if updating ('address') then
    emp_tab_pkg.insert_value(null,null,null,null,null,:new.address,null,null,null,null,null,null);
    emp_tab_pkg.check_field := 5;
  end if;

  end if;

165
if updating('sex') then
    emp_tab_pkg.insert_value(null, null, null, null, null, null, :new.sex, null, null, null, null);
    emp_tab_pkg.check_field := 6;
end if;

if updating('salary') then
    emp_tab_pkg.insert_value(mew.empno, :old.empno, null, null, mew.bdate, null, null, :new.salary, :old.salary, mew.ssn, mew.dno);
    emp_tab_pkg.check_field := 7;
end if;

if updating('ssn') then
    emp_tab_pkg.insert_value(null, null, null, null, null, null, null, null, null, mew.ssn, null);
    emp_tab_pkg.check_field := 8;
end if;

if updating('dno') then
    emp_tab_pkg.insert_value(null, null, null, null, null, null, null, null, null, null, mew.dno);
    emp_tab_pkg.check_field := 9;
end if;
end emp_aur;

CREATE OR REPLACE TRIGGER "COMPANY1"."EMP_AUS"
AFTER UPDATE ON "COMPANY1"."EMPLOYEE"
declare
    old_emp_ssn employee.empno%type;
    
    -- Cursor keeps employees who have the same ssn as
    -- the updated employee in EMPLOYEE table.
    cursor get_ssn is
        select empno
        from employee
        where ssn = emp_cons_pkg. oldmmpno;
    begin
        emp_tab_pkg.set_iterator;
        while emp_tab_pkg.more_in_table loop
            -- check_field = 1: update empno field must be unique and
cannot be null.
            if emp_tab_pkg.check_field = 1 then
                emp_cons_pkg.enforce_pk_empno_upd(emp_tab_pkg.next_rec);
            end if;
            -- check_field = 2: update fname field cannot be null.
            if emp_tab_pkg.check_field = 2 then
                emp_cons_pkg.check_fname_upd(emp_tab_pkg.next_rec);
            end if;
            -- check_field = 3: update lname field cannot be null.
if emp_tab_pkg.check_field = 3 then
    emp_cons_pkg.check_lname_upd(emp_tab_pkg.next_rec);
end if;

-- check_field = 4: update birth date field cannot be null.
if emp_tab_pkg.check_field = 4 then
    emp_cons_pkg.check_bdate_upd(emp_tab_pkg.next_rec);
end if;

-- check_field = 5: update address field cannot be null.
if emp_tab_pkg.check_field = 5 then
    emp_cons_pkg.check_address_upd(emp_tab_pkg.next_rec);
end if;

-- check_field = 6: update sex field can be either f or m.
if emp_tab_pkg.check_field = 6 then
    emp_cons_pkg.enforce_sex_upd(emp_tab_pkg.next_rec);
end if;

-- check_field = 7: update salary
if emp_tab_pkg.check_field = 7 then
    emp_cons_pkg.enforce_sal_upd(emp_tab_pkg.next_rec);
end if;

-- check = 8: update ssn field must be an employee in
EMPLOYEE table.
if emp_tab_pkg.check_field = 8 then
    emp_cons_pkg.enforce_fk_ssn_upd(emp_tab_pkg.next_rec);
end if;

-- check = 9: update dno field must be a department in
DEPARTMENT table.
if emp_tab_pkg.check_field = 9 then
    emp_cons_pkg.enforce_fk_dno_upd(emp_tab_pkg.next_rec);
end if;
end loop;

-- Update ssn when the updating employee is also a supervisor.
if emp_cons_pkg.old_empno != emp_cons_pkg.new_empno then
    open get_ssn;
    loop
        fetch get_ssn into bld_emp_ssn;
        exit when get_ssn%notfound;
        update employee
            set ssn = emp_cons_pkg.new_empno
            where ssn = emp_cons_pkg.old_empno;
    end loop;
end if;
end emp_aus;
LOCATION:

CREATE OR REPLACE TRIGGER "COMPANY1"."LOC_BIS"
BEFORE INSERT ON "COMPANY1"."LOCATION"
begin
    loc_tab_pkg.clear_table;
end loc_bis;

CREATE OR REPLACE TRIGGER "COMPANY1"."LOC_AIR"
AFTER INSERT ON "COMPANY1"."LOCATION"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    loc_tab_pkg.insert_value(:new.dno, null, :new.dlocation,
    null);
end loc_air;

CREATE OR REPLACE TRIGGER "COMPANY1"."LOC_AIS"
AFTER INSERT ON "COMPANY1"."LOCATION"
begin
    loc_tab_pkg.set_iterator;
    while loc_tab_pkg.more_in_table loop
        loc_cons_pkg.enforce_pk_loc(loc_tab_pkg.next_rec);
    end loop;
end loc_ais;

CREATE OR REPLACE TRIGGER "COMPANY1"."LOC_BUS"
BEFORE UPDATE ON "COMPANY1"."LOCATION"
begin
    loc_tab_pkg.clear_table;
end loc_bus;

CREATE OR REPLACE TRIGGER "COMPANY1"."LOC_AUR"
BEFORE UPDATE OF "DLOCATION", "DNO" ON "COMPANY1"."LOCATION"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    loc_tab_pkg.insert_value(:new.dno, :old.dno, :new.dlocation,
    :old.dlocation);
end loc_aur;

CREATE OR REPLACE TRIGGER "COMPANY1"."LOC_AUS"
AFTER UPDATE ON "COMPANY1"."LOCATION"
begin
    loc_tab_pkg.set_iterator;
    while loc_tab_pkg.more_in_table loop
        loc_cons_pkg.enforce_pk_loc(loc_tab_pkg.next_rec);
    end loop;
end loc_aus;
PROJECT:

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJ_BDR"
BEFORE DELETE ON "COMPANY1"."PROJECT"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    proj_cons_pkg.project_del(:old.pnumber, :old.pname, :old.dno);
end proj_bdr;

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJ_BIR"
BEFORE INSERT ON "COMPANY1"."PROJECT"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    proj_cons_pkg.enforce_pk_pnum_ins(:new.pnumber);
    proj_cons_pkg.enforce_pk_pname_ins(:new.pname);
    proj_cons_pkg.enforce_fk_dno_ins(:new.dno);
end proj_bir;

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJ_BUS"
BEFORE UPDATE ON "COMPANY1"."PROJECT"
begin
    proj_tab_pkg.clear_table;
end proj_bus;

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJ_AUR"
AFTER UPDATE OF "DNO", "PNAME", "PNUMBER" ON "COMPANY1"."PROJECT"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    if updating ('pnumber') then
        proj_tab_pkg.insert_value(:new.pnumber, :old.pnumber, null, null);
        proj_tab_pkg.check_field := 1;
    end if;

    if updating ('dno') then
        proj_tab_pkg.insert_value(null, null, :new.dno, null);
        proj_tab_pkg.check_field := 3;
    end if;

    if updating ('pname') then
        proj_tab_pkg.insert_value(null, null, null, :new.pname);
        proj_tab_pkg.check_field := 4;
    end if;
end proj_aur;

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJ_AUS"
AFTER UPDATE ON "COMPANY1"."PROJECT"
begin
    proj_tab_pkg.set_iterator;
end proj_aus;
while proj_tab_pkg.more_in_table loop
    if proj_tab_pkg.check_field = 1 then
        proj_cons_pkg.enforce_pk_pnum_upd(proj_tab_pkg.next_rec);
        end if;
        if proj_tab_pkg.check_field = 3 then
            proj_cons_pkg.enforce_fk_dno_upd(proj_tab_pkg.next_rec);
            end if;
            if proj_tab_pkg.check_field = 4 then
                proj_cons_pkg.enforce_uk_pname_upd(proj_tab_pkg.next_rec);
            end if;
        end loop;
end proj_aus;
WORKS_ON:

CREATE OR REPLACE TRIGGER "COMPANY1"."WORKS_ON_BIR"
BEFORE INSERT ON "COMPANY1"."WORKS_ON"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    works_on_cons_pkg.enforce_fk_empno(:new.empno);
    works_on_cons_pkg.enforce_fk_pnum(:new.pnumber);
    works_on_cons_pkg.enforce_pk_work_ins(:new.empno, :new.pnumber);
    works_on_cons_pkg.check_hours(:new.hours);
    works_on_cons_pkg.enforce_work_ins(:new.empno, :new.pnumber);
end works_on_bir;

CREATE OR REPLACE TRIGGER "COMPANY1"."WORKS_ON_BUS"
BEFORE UPDATE ON "COMPANY1"."WORKS_ON"
begin
    works_on_tab_pkg.clear_table;
end works_on_bus;

CREATE OR REPLACE TRIGGER "COMPANY1"."WORKS_ON_AUR"
BEFORE UPDATE OF "EMPNO", "PNUMBER" ON "COMPANY1"."WORKS_ON"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    works_on_tab_pkg.insert_value(:new.empno, :old.empno, :new.pnumber, :old.pnumber);
end works_on_aur;

CREATE OR REPLACE TRIGGER "COMPANY1"."WORKS_ON_AUS"
AFTER UPDATE ON "COMPANY1"."WORKS_ON"
begin
    works_on_tab_pkg.set_iterator;
    while works_on_tab_pkg.more_in_table loop
        works_on_cons_pkg.enforce_pk_work_upd((works_on_tab_pkg.next_rec);
    end loop;
end works_on_aus;
PROJECT_INFO:

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJECT_INFO_DEL"
INSTEAD OF DELETE ON "COMPANY1"."PROJECT_INFO"
declare count1 number;
count2 number;
begin /* Check the deleted department is in DEPARTMENT table or not.*/
select count(*) into count1
from department
where dno = :old.dno;
/* Check the deleted project is in the PROJECT table or not.*/
select count(*) into count2
from project
where pnumber = :old.pnumber;
/* Delete the project that belongs the department. */
if (count1 > 0) and (count2 > 0) then
  delete from project
  where pnumber = :old.pnumber;
end if;
end project_info_del;

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJECT_INFO_INS"
INSTEAD OF INSERT ON "COMPANY1"."PROJECT_INFO"
declare count1 number;
count2 number;
count3 number;
count4 number;
department.mgrssn%type dept_default;
begin
-- Check constraints when insert the project number.
if :new.pnumber is null then
  raise_application_error(project_errors.pnum_nn,
project_errors.pnum_nn_msg );
else
  -- Find the number of projects that have the same project
  number
  select count(*) into count1
  from project
  where pnumber = :new.pnumber;
  -- Check a primary key constraint on the project number.
  if count1 = 0 then
    -- Check a not null constraint when insert the name of
  project.
    if :new.pname is null then
      raise_application_error(project_errors.pname_nn,
project_errors.pname_nn_msg);
    else
      -- Check a unique key constraint when insert the
      name of project.
      "COMPANY1"."PROJECT_INFO_INS"
select count(*) into countt4
from project
where upper(pname) = upper(:new.pname);

if countt4 > 0 then
  raise_application_error(project_errors.pname_uk,
project_errors.pname_uk_msg);
  end if;
else
  end if;
  -- Display an error message if insert an existing
  project.
  raise_application_error(project_errors.proj_exists,
project_errors.proj_exists_msg);
  end if;
end if; /* end checking project number. */

-- Check constraints on the department.
-- Display an error message if the department number is null.
if :new.dno is null then
  raise_application_error(project_errors.proj_control_dno,
project_errors.proj_control_dno_msg);
else
  -- Find the number of departments that have the same
department number
  select count(*) into countt1
  from department
  where dno = :new.dno;

  -- Insert the new department into the DEPARTMENT table
  -- and assign the default manager, mgrssn = '999999999',
  for the new department.
  if countt1 = 0 then
    select count(*) into countt4
    from department
    where upper(dname) = upper(:new.dname);

    -- The name of new department cannot be the same as that
    of an existing one.
    if countt4 > 0 then
      raise_application_error(proj_errors.dname_uk,
proj_errors.dname_uk_msg);
    else
      insert into department(dno, dname, mgrssn)
      values(:new.dno, :new.dname, '999999999');
      end if;
    else
      -- Insert a project with an existing department, check
      the name of department.
      -- Find the number of department that has the same
department number
      -- and department's name as the inserted one.
      select count(*) into countt3
      from department
      where upper(dname) = upper(:new.dname)
      and dno = :new.dno;

      173
-- If insert an existing department, the name of
department can be omitted (null).
-- If the name of department is given, it must be the
same as an existing one.
if (countt3 = 0) and (:new.dname is not null) then
  raise_application_error(project_errors.dno_exist,
  project_errors.dno_exist_msg);
end if;
end if;
end if; /* end checking department.*/
insert into project(pnumber, pname, dno)
values(:new.pnumber, :new.pname, :new.dno);
end project_info_ins;

CREATE OR REPLACE TRIGGER "COMPANY1"."PROJECT_INFO_UPD"
INSTEAD OF UPDATE ON "COMPANY1"."PROJECT_INFO"
declare
  counttl number;
  countt2 number;
begin
  if updating('pnumber') then
    -- Display an error if update project number to a null value.
    if :new.pnumber is null then
      raise_application_error(project_errors.pnum_nn,
      project_errors.pnum_nn_msg);
    else
      -- Display an error if update project number to an existing
      one.
      select count(pnumber) into counttl
      from project
      where pnumber = :new.pnumber;
      -- If no violation occurs, update the project number in the
      PROJECT table.
      if countt1 = 0 then
        update project
        set pnumber = :new.pnumber
        where pnumber = :old.pnumber;
      else
        raise_application_error(project_errors.pnum_exist,
        project_errors.pnum_exist_msg);
      end if;
    end if;
  end if; /* End of updating pnumber. */

  if updating('pname') then
    -- Display an error if update a project name to a null value.
    if :new.pname is null then
      raise_application_error(project_errors.pname_nn,
      project_errors.pname_nn_msg);
    else
      -- Display an error if update a project name to an exist
      one.
      select count(*) into counttl
      from project

where upper(pname) = upper(:new.pname);

if counttl = 0 then
    update project
    set pname = :new.pname
    where pnumber = :old.pnumber;
else
    raise_application_error(project_errors.pname_uk, project_errors.pname_uk_msg);
end if;
end if; /* End of updating pname. */

if updating('dno') then
    if :new.dno is null then
        raise_application_error(project_errors.proj_control_dno, project_errors.proj_control_dno_msg);
    else
        -- Check the department does exist.
        select count(*) into countt2
        from department
        where dno = :new.dno;

        -- Update the department that controls a project.
        if countt2 > 0 then
            update project
            set dno = :new.dno
            where pnumber = :old.pnumber;
        else
            -- Display an error message if the department is not updated to an existing one.
            raise_application_error(project_errors.proj_control_dno, project_errors.proj_control_dno_msg);
        end if;
    end if;
end if; /* End of updating dno. */

if updating('dname') then
    raise_application_error(project_errors.dname_is_used, project_errors.dname_is_used_msg);
end if;
end project_info_upd;
ERROR PACKAGES:

/* Error package for the EMPLOYEE and DEPARTMENT tables. */
CREATE OR REPLACE PACKAGE PROJ_ERRORS is
-- Error codes for EMPLOYEE table.
empno_nn constant number := -20100;
fname_nn constant number := -20101;
lname_nn constant number := -20102;
bdate_nn constant number := -20103;
address_nn constant number := -20104;
salary_nn constant number := -20105;
empno_uk constant number := -20106;
sex_ck constant number := -20107;
salary_ck constant number := -20108;
ssn_fk constant number := -20110;
dno_fk constant number := -20111;
sal_over_ssn number := -20116;
sal_higher_limit number := -20117;
sal_age_limit number := -20118;
age_limit number := -20119;
sal_over_mgr number := -20120;

-- Error messages for EMPLOYEE table.
empno_nn_msg varchar2(50) :=
  'Employee number cannot be null.';
fname_nn_msg varchar2(50) :=
  'Employee first name cannot be null.';
lname_nn_msg varchar2(50) :=
  'Employee last name cannot be null.';
bdate_nn_msg varchar2(50) :=
  'Employee birthdate cannot be null.';
address_nn_msg varchar2(50) :=
  'Employee address cannot be null.';
salary_nn_msg varchar2(50) :=
  'The salary of employee cannot be null.';
empno_uk_msg varchar2(50) not null :=
  'Employee number must be unique.';
sex_ck_msg varchar2(100) not null :=
  'The gender of employee can be either M male or F female.';
salary_ck_msg varchar2(50) not null :=
  'The salary of employee cannot be less than zero.';
ssn_fk_msg varchar2(100) not null :=
  'Supervisor must be an existing employee in EMPLOYEE table or null value.';
dno_fk_msg varchar2(100) not null :=
  'Department number must be an existing department in DEPARTMENT table or null
  sal_over_ssn_msg varchar2(100) not null :=
  'The salary of employee cannot be higher than that of his/her supervisor.';
sal_higher_limit_msg varchar2(70) not null :=
  'The salary of an employee, not manager, cannot increase more than 15%.';
sal_age_limit_msg varchar2(70) not null :=
  'Employee who is more than 60 years old cannot get a raise in salary.';
age_limit_msg varchar2(70) not null :=

176
The age of employee must be between 18 and 65 years old.'; 
sal_over_mgr_msg varchar2(100) not null := 
'The salary of employee cannot be higher than that of the manager 
of department.';

-- Error codes in DEPARTMENT table.
dno_nn constant number := -20130;
dno_uk constant number := -20131;
mgrssn_fk constant number := -20132;
default_dno constant number := -20133;
dname_uk constant number := -20134;
dname_nn constant number := -20135;

-- Error messages in DEPARTMENT table.
dno_nn_msg varchar2(50) not null := 
'Department number cannot be null.';
dno_uk_msg varchar2(50) not null := 
'Department number must be unique.';
mgrssn_fk_msg varchar2(100) not null := 
'The manager of department must be an existing employee.';
default_dno_msg varchar2(100) not null := 
'A default department (department number = 10) cannot be deleted.';
dname_uk_msg varchar2(50) not null := 
'The name of department already exists.';
dname_nn_msg varchar2(50) not null := 
'The name of department cannot be null.';
end proj_errors;

/* Error package for the LOCATION table. */
CREATE OR REPLACE PACKAGE loc_errors is
dlocation_nn constant number := -20500;
dno_dloc_uk constant number := -20501;

dlocation_nn_msg varchar2(50) not null :=
'The location of department cannot be null.';
dno_dloc_uk_msg varchar2(100) not null :=
'The department and location must be unique.';
end loc_errors;

/* Error package for the PROJECT table and PROJECT_INFO view. */
CREATE OR REPLACE PACKAGE PROJECT_ERRORS is

-- Error codes in PROJECT table (-20400 to -20499).
pnum_nn constant number := -20400;
pnum_uk constant number := -20401;
pname_dno_fk constant number := -20402;
pname_uk constant number := -20403;
proj_control_dno number := -20404;
pname_nn constant number := -20405;
proj_exists constant number := -20406;

-- Error messages in PROJECT table.
pnum_nn_msg varchar2(50) not null :=
'Project number cannot be null.';
'pnum__uk_msg varchar2(50) not null :=
'Project number must be unique.';
'proj_dno_fk_msg varchar2(100) not null :=
'A project must be controlled by a department.';
'pname__uk_msg varchar2(50) not null :=
'The name of project already exists.';
'proj_control_dno_msg varchar2(100) not null :=
'A project must be controlled by an existing department.';
'pname_nn_msg varchar2(50) not null :=
'The name of project cannot be null.';
'proj_exists_msg varchar2(50) not null :=
'The project already exists in the table.';
-- Error codes for the PROJECT_INFO view.
dno_exist constant number := -20407;
pnum_exist constant number := -20408;
dname_is__used constant number := -20409;
-- Error message in the PROJECT_INFO view.
dno_exist_msg varchar2(100) not null :=
'The department already exists. The name of department cannot be
different from that of an existing one.';
pnum_exist_msg varchar2(100) not null :=
'Project number cannot be updated to an existing project.';
dname_is_used_msg varchar2(150) not null :=
'The name of department does not allow to update since some
projects may be controlled by that department.';
end project_errors;

/* Error package for the WORKS_ON table. */
CREATE OR REPLACE PACKAGE WORKS_ON_ERRORS IS
-- Error codes in WORKS_ON table (-20300 to -20399)
empno_fk constant number := -20300;
pnum_fk constant number := -20301;
empno_pnum__uk constant number := -20302;
hours_ck constant number := -20303;
proj_limit constant number := -20304;
-- Error message in WORKS_ON table.
empno_fk_msg varchar2(700) not null :=
'Employee number must be an employee in EMPLOYEE table.';
pnum_fk_msg varchar2(70) not null :=
'Project number must be a project in PROJECT table.';
empno_pnum__uk_msg varchar2(70) not null :=
'Employee and Project number must be unique.';
hours_ck_msg varchar2(70) not null :=
'Employee work hours cannot be less than zero';
proj_limit_msg varchar2(70) not null :=
'Employee, not manager, cannot work more than 10 projects.';
end works_on_errors;

CREATE OR REPLACE PACKAGE DEPT_TAB_PKG is
check_field number := 0; -- check which field is updated.
type rec_type is record(
  new_dno department.dno%type,
  old_dno department.dno%type,
  new_mgrsn department.mgrsn%type,
  new_dname department.dname%type);
procedure clear_table;
procedure set_iterator;
function more_in_table return boolean;
function next_rec return rec_type;
procedure insert_value(new_dno department.dno%type,
  old_dno department.dno%type,
  new_mgrsn department.mgrsn%type,
  new_dname department.dname%type);
end dept_tab_pkg;

CREATE OR REPLACE PACKAGE BODY DEPT_TAB_PKG is
type dno_tab_type is table of department.dno%type
  index by binary_integer;
type mgrsn_tab_type is table of department.mgrsn%type
  index by binary_integer;
type dname_tab_type is table of department.dname%type
  index by binary_integer;
new_dno_tab dno_tab_type;
old_dno_tab dno_tab_type;
mgrsn_tab mgrsn_tab_type;
dname_tab dname_tab_type;
table_count number;
table_iterator number := 0;

-- Procedure clear_table: Clear the counter of table to zero.
procedure clear_table is
begin
  table_count := 0;
end clear_table;

-- Procedure set_iterator: Assign the total number of rows
-- after manipulating to the table_iterator variable.
procedure set_iterator is
begin
  table_iterator := table_count;
end set_iterator;

-- Function more_in_table: Check data exists in the table.
function more_in_table return boolean is
begin
  return (table_iterator > 0);
end more_in_table;
end more_in_table;

-- Function next_rec: Return the data in the table.
function next_rec return rec_type is rec rec_type;
begint
rec.new_dno := new_dno_tab(table_iterator);
rec.old_dno := old_dno_tab(table_iterator);
rec.new_mgrssn := mgrssn_tab(table_iterator);
rec.new_dname := dname_tab(table_iterator);
table_iterator := table_iterator - 1;
return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure insert_value(new_dno department.dno%type,
old_dno department.dno%type,
new_mgrssn department.mgrssn%type,
new_dname department.dname%type) is
begin
  table_count := table_count + 1;
  new_dno_tab(table_count) := new_dno;
  old_dno_tab(table_count) := old_dno;
  mgrssn_tab(table_count) := new_mgrssn;
  dname_tab(table_count) := new_dname;
end insert_value;
end dept_tab_pkg;
CREATE OR REPLACE PACKAGE `DEL_DNO_TAB_PKG` is
  type rec_type is record(
    new_dno department.dno%type,
    old_dno department.dno%type,
    old_dname department.dname%type);
  procedure clear_table;
  procedure set_iterator;
  function more_in_table return boolean;
  function next_rec return rec_type;
  procedure insert_value(new_dno department.dno%type,
                          old_dno department.dno%type,
                          old_dname department.dname%type);
end del_dno_tab_pkg;

CREATE OR REPLACE PACKAGE BODY `DEL_DNO_TAB_PKG` is
  type dno_tab_type is table of department.dno%type
    index by binary_integer;
  type dname_tab_type is table of department.dname%type
    index by binary_integer;

  new_dno_tab dno_tab_type;
  old_dno_tab dno_tab_type;
  old_dname_tab dname_tab_type;
  table_count number;
  table_iterator number := 0;

  -- Procedure clear_table: Clear the counter of table to zero.
  procedure clear_table is begin
    table_count := 0;
  end clear_table;

  -- Procedure set_iterator: Assign the total number of rows
  -- after manipulating to the table_iterator variable.
  procedure set_iterator is begin
    table_iterator := table_count;
  end set_iterator;

  -- Function more_in_table. Check data exists in the table.
  function more_in_table return boolean is begin
    return (table_iterator > 0);
  end more_in_table;

  -- Function next_rec: Return the data in the table.
  function next_rec return rec_type is rec rec_type;
  begin
    rec.new_dno := new_dno_tab(table_iterator);
    rec.old_dno := old_dno_tab(table_iterator);
    rec.old_dname := old_dname_tab(table_iterator);
    table_iterator := table_iterator - 1;
    return rec;
  end next_rec;

  -- Procedure insert_value: Insert data into the table
procedure insert_value(new_dno department.dno%type,
    old_dno department.dno%type,
    old_dname department.dname%type) is
begin
    table_count := table_count + 1;
    new_dno_tab(table_count) := new_dno;
    old_dno_tab(table_count) := old_dno;
    old_dname_tab(table_count) := old_dname;
end insert_value;
end del_dno_tab_pkg;
CREATE OR REPLACE PACKAGE DEPT_CONS_PKG is
  procedure dno_num(dept_dno department.dno%type,
    rownum1 out number);
  procedure mgrssn_num(dept_mgrssn department.mgrssn%type,
    rownum2 out number);
  procedure dno_emp(dept_dno department.dno%type,
    rownum3 out number);
  procedure dno_proj(dept_dno department.dno%type,
    rownum4 out number);
  procedure dno_loc(dept_dloc department.dno%type,
    rownum5 out number);
  procedure dname_num(dept_dname department.dname%type,
    rownum6 out number);
  procedure enforce_pk_dno_ins(new_dno department.dno%type);
  procedure enforce_pk_dno_upd(rec dept_tab_pkg.rec_type);
  procedure enforce_uk_dname_ins(new_dname department.dname%type);
  procedure enforce_uk_dname_upd(rec dept_tab_pkg.rec_type);
  procedure dept_del(old_dno department.dno%type,
    old_dname department.dname%type);
  procedure audit_proj(old_dno department.dno%type,
    old_dname department.dname%type);
  procedure dept_upd_cascade(rec dept_tab_pkg.rec_type);
  procedure enforce_fk_mgr_ins(new_mgrssn department.mgrssn%type);
  procedure enforce_fk_mgr_upd(rec dept_tab_pkg.rec_type);
end dept_cons_pkg;

CREATE OR REPLACE PACKAGE BODY DEPT_CONS_PKG is
  -- Procedure Name: dno_num
  -- Purpose: Find the number of departments that have the same number
  -- as an inserted one.
  procedure dno_num(dept_dno department.dno%type,
    rownum1 out number) as
    cursor get_dno_row(dept_dno department.dno%type) is
      select count(*)
      from department
      where dno = dept_dno;
    begin
      open get_dno_row(dept_dno);
      fetch get_dno_row into rownum1;
      close get_dno_row;
    end dno_num;

  -- Procedure Name: mgrssn_num
  -- Purpose: Find the number of employees who have the same employee
  -- number as the department’s manager.
  procedure mgrssn_num(dept_mgrssn department.mgrssn%type,
    rownum2 out number) as
    cursor get_mgrssn_row(dept_mgrssn department.mgrssn%type) is
      select count(*)
      from employee
      where empno = dept_mgrssn;
    begin
      open get_mgrssn_row(dept_mgrssn);
      fetch get_mgrssn_row into rownum2;
      close get_mgrssn_row;
    end mgrssn_num;
-- Procedure Name: dno_emp
-- Purpose: Find the number of employees who work on the deleted department.
procedure dno_emp (dept_dno department.dno%type,
                   rownum3 out number) as
    cursor get_emp_row (dept_dno department.dno%type) is
        select count(*)
        from employee
        where dno = dept_dno;
begin
    open get_emp_row (dept_dno);
    fetch get_emp_row into rownum3;
    close get_emp_row;
end dno_emp;

-- Procedure Name: dno_proj
-- Purpose: Find the number of projects that are controlled by the deleted department.
procedure dno_proj (dept_dno department.dno%type,
                     rownum4 out number) as
    cursor get_proj_row (dept_dno department.dno%type) is
        select count(*)
        from project
        where dno = dept_dno;
begin
    open get_proj_row (dept_dno);
    fetch get_proj_row into rownum4;
    close get_proj_row;
end dno_proj;

-- Procedure Name: dno_loc
-- Purpose: Find the number of locations of the deleted department.
procedure dno_loc (dept_dloc department.dno%type,
                   rownum5 out number) as
    cursor get_dloc_row (dept_dloc department.dno%type) is
        select count(*)
        from location
        where dno = dept_dloc;
begin
    open get_dloc_row (dept_dloc);
    fetch get_dloc_row into rownum5;
    close get_dloc_row;
end dno_loc;

-- Procedure Name: dname_num
-- Purpose: Find the number of departments that have the same name as an inserted one.
procedure dname_num (dept_dname department.dname%type,
                      rownum6 out number) as
    cursor get_dname_row (dept_dname department.dname%type) is
        select count(*)
        from department
        where upper(dname) = upper(dept_dname);
begin
    open get_dname_row (dept_dname);

fetch get_dname_row into rownum6;
close get_dname_row;
end dname_num;

-- Procedure Name: enforce_pk_dno_ins
-- Purpose: Enforce a primary key constraint when insert the department number.
procedure enforce_pk_dno_ins(new_dno department.dno%type)
as total number;
begin
-- Display an error message if the department number is null.
if new_dno is null then
  raise_application_error
    (proj_errors.dno_nn, proj_errors.dno_nn_msg);
else
-- Display an error message if the department number already exists.
  dno_num(new_dno, total);
  if total > 0 then
    raise_application_error(proj_errors.dno_uk,
                             proj_errors.dno_uk_msg);
  end if;
end if;
end if;
end enforce_pk_dno_ins;

-- Procedure Name: enforce_pk_dno_upd
-- Purpose: Enforce a primary key constraint when update the department number.
procedure enforce_pk_dno_upd(rec dept_tab_pkg.rec_type)
is countt number;
begin
-- Find the number of department that has the same as the inserted one.
  select count(*) into countt
    from department
    where dno = rec.new_dno;

-- Display error message if the department is updated to a null value.
  if rec.new_dno is null then
    raise_application_error
      (proj_errors.dno_nn, proj_errors.dno_nn_msg);
  else
-- Display error message if the department is updated to an existing one.
    if countt > 1 then
      raise_application_error(proj_errors.dno_uk,
                               proj_errors.dno_uk_msg);
    else
      -- Update cascade department number.
      dept_upd_cascade(rec);
    end if;
  end if;
end if;
end enforce_pk_dno_upd;

-- Procedure Name: enforce_uk_dname_ins
-- Purpose: Enforce not null and unique key constraints when insert
the department.
procedure enforce uk_dname_ins(new_dname department.dname%type)
as total number;
begi
  -- Display an error message if the department name is null.
  if new_dname is null then
    raise_application_error
    (proj_errors.dname_nn, proj_errors.dname_nn_msg);
  else
    -- Display an error message if the department name already
    -- exists.
    dname_num(new_dname, total);
    if total > 0 then
      raise_application_error(proj_errors.dname_uk,
      proj Errors.dname_uk_msg);
    end if;
  end if;
end enforce uk_dname_ins;

-- Procedure Name: enforce uk_dname_upd
-- Purpose: Enforce not null and unique key constraint when update the
department.
procedure enforce uk_dname_upd(rec dept_tab_pkg.rec_type) as
count number;
begi
  -- Find the number of department that has the same name as the
  -- inserted one.
  select count(*) into count
  from department
  where upper(dname) = upper(rec.new_dname);
  -- Display error message if the department name is updated to a
  -- null value.
  if rec.new_dname is null then
    raise_application_error
    (proj_errors.dname_nn, proj_errors.dname_nn_msg);
  else
    -- Display error message if the department's name is updated
to an existing one.
    if count > 1 then
      raise_application_error(proj_errors.dname_uk,
      proj Errors.dname_uk_msg);
    end if;
  end if;
end enforce uk_dname_upd;

-- Procedure Name: dept_del
-- Purpose: Enforce a referential integrity constraint when delete a
department.
procedure dept_del(old_dno department.dno%type,
    old_dname department.dname%type) as
cursor emp_of_dept_cur(old_dno department.dno%type) is
select empno
from employee
where dno = old_dno;
count number;
begin
    -- If the deleted department has employees working on it,
    -- set the department of those employees to null.
    dno_emp(old_dno, countt);

    if countt > 0 then
        -- keep the information of employee department in
        -- department_aud table.
        open emp_of_dept_cur(old_dno);
        loop
            fetch emp_of_dept_cur into emp_of_dno;
            exit when emp_of_dept_cur%notfound;
            insert into department_aud
                values(old_dno, old_dname, emp_of_dno, sysdate,
                to_char(sysdate, 'hh24:mi:ss'));
        end loop;
    end if;

    update employee
        set dno = null
        where dno = old_dno;
end if;

    -- The locations of the deleted department are also deleted
    -- when a department is deleted from the table.
    dno_loc(old_dno, countt);

    if countt > 0 then
        delete from location
        where dno = old_dno;
    end if;
end dept_del;

-- Procedure Name: audit_proj
-- Purpose: Keep the project information that belongs to the delete
department.
procedure audit_proj(old_dno department.dno%type,
old_dname department.dname%type) as
cursor proj_default_cur(old_dno department.dno%type) is
    select pnumber, pname
    from project
    where dno = old_dno;
proj_default proj_default_cur%rowtype;
begin
    open proj_default_cur(old_dno);
    loop
        fetch proj_default_cur into proj_default;
        exit when proj_default_cur%notfound;
        insert into dept_of_proj_aud
            values(proj_default.pnumber, proj_default.pname,
            old_dno, old_dname, sysdate, to_char(sysdate, 'hh24:mi:ss'));
    end loop;
end audit_proj;

-- Procedure Name: dept_upd_cascade
-- Purpose: An update cascade to the EMPLOYEE, PROJECT, LOCATION
tables
-- when update a department number
procedure dept_upd_cascade(rec dept_tab_pkg.rec_type)
is countt integer;

begin
  -- If this department has employees working on it,
  -- update the department number of these employees to the new one.
  select count(*) into countt
  from employee
  where dno = rec.old_dno;

  if countt > 0 then
    update employee
    set dno = rec.new_dno
    where dno = rec.old_dno;
  end if;

  -- If this department controls projects,
  -- set the department number in the PROJECT table to the new one.
  select count(*) into countt
  from project
  where dno = rec.old_dno;

  if countt > 0 then
    update project
    set dno = rec.new_dno
    where dno = rec.old_dno;
  end if;

  -- If the department is updated, update the department in the
  LOCATION table.
  select count(*) into countt
  from location
  where dno = rec.old_dno;

  if countt > 0 then
    update location
    set dno = rec.new_dno
    where dno = rec.old_dno;
  end if;
end dept_upd_cascade;

-- Procedure Name: enforce_fk_mgr_ins
-- Purpose: Enforce a referential integrity constraint when insert a
manager of department.
-- A manager of department must be an existing employee.
procedure enforce_fk_mgr_ins(new_mgrssn department.mgrssn%type)
as total number;
begin
  -- Display an error message if the manager of department does not
  exist in the EMPLOYEE table.
  if new_mgrssn is not null then
    mgrssn_num(new_mgrssn, total);

    if total < 1 then
      raise_application_error(proj_errors.mgrssn_fk,
                               proj_errors.mgrssn_fk_msg);
    end if;
  end if;
end if;
end enforce_fk_mgr_ins;

-- Procedure Name: enforce_fk_mgr_upd
-- Purpose: Enforce a referential integrity constraint when update the
-- manager of department.
-- The manager of department must be updated to an existing employee.
procedure enforce_fk_mgr_upd(rec dept_tab_pkg.rec_type)
is countt number;
begin
  if rec.new_mgrssn is null then
    return;
  end if;

  select count(*) into countt
  from employee
  where rec.new_mgrssn = empno;

  -- Display an error message if the manager of department is not
  -- updated to an existing employee.
  if (rec.new_mgrssn is not null) and (countt < 1) then
    raise_application_error
      (proj_errors.mgrssn_fk, proj_errors.mgrssn_fk_msg);
  end if;
end enforce_fk_mgr_upd;
end dept_cons_pkg;
CREATE OR REPLACE PACKAGE EMP_TAB_PKG is

check_field number := 0; -- check which field is updated.
type rec_type is record(
    empno employee.empno%type,
    old_empno employee.empno%type,
    new_fname employee.fname%type,
    new_lname employee.lname%type,
    new_bdate employee.bdate%type,
    new_address employee.address%type,
    new_sex employee.sex%type,
    new_salary employee.salary%type,
    old_salary employee.salary%type,
    new_ssn employee.ssn%type,
    new_dno employee.dno%type);

procedure clear_table;
procedure set_iterator;
function more_in_table return boolean;
function next_rec return rec_type;
procedure insert_value(empno employee.empno%type,
    old_empno employee.empno%type,
    new_fname employee.fname%type,
    new_lname employee.lname%type,
    new_bdate employee.bdate%type,
    new_address employee.address%type,
    new_sex employee.sex%type,
    new_salary employee.salary%type,
    old_salary employee.salary%type,
    new_ssn employee.ssn%type,
    new_dno employee.dno%type);

end emp_tab_pkg;

CREATE OR REPLACE PACKAGE BODY EMP_TAB_PKG is

type emp_tab_type is table of employee.empno%type
    index by binary_integer;
type fname_tab_type is table of employee.fname%type
    index by binary_integer;
type lname_tab_type is table of employee.lname%type
    index by binary_integer;
type bdate_tab_type is table of employee.bdate%type
    index by binary_integer;
type address_tab_type is table of employee.address%type
    index by binary_integer;
type sex_tab_type is table of employee.sex%type
    index by binary_integer;
type salary_tab_type is table of employee.salary%type
    index by binary_integer;
type ssn_tab_type is table of employee.ssn%type
    index by binary_integer;
type dno_tab_type is table of employee.dno%type
    index by binary_integer;

emp_tab emp_tab_type;
old_emp_tab emp_tab_type;
**fname_tab** fname_tab_type;
**lname_tab** lname_tab_type;
**bdate_tab** bdate_tab_type;
**address_tab** address_tab_type;
**sex_tab** sex_tab_type;
**salary_tab** salary_tab_type;
**old_sal_tab** salary_tab_type;
**ssn_tab** ssn_tab_type;
**dno_tab** dno_tab_type;
**table_count** number;
**table_iterator** number := 0;

-- Procedure clear_table: Clear the counter of table to zero.
procedure **clear_table** is
begin
    table_count := 0;
end clear_table;

-- Procedure set_iterator: Assign the total number of rows
-- after manipulating to the table_iterator violation.
procedure **set_iterator** is
begin
    table_iterator := table_count;
end set_iterator;

-- Function more_in_table: Check data exists in the table.
function **more_in_table** return boolean is
begin
    return (table_iterator > 0);
end more_in_table;

-- Function next_rec: Return the data in the table.
function **next_rec** return rec_type is rec rec_type;
begin
    rec empno := emp_tab(table_iterator);
    rec old_empno := old_emp_tab(table_iterator);
    rec new_fname := fname_tab(table_iterator);
    rec new_lname := lname_tab(table_iterator);
    rec new_bdate := bdate_tab(table_iterator);
    rec new_address := address_tab(table_iterator);
    rec new_salary := salary_tab(table_iterator);
    rec new_ssn := ssn_tab(table_iterator);
    rec new_dno := dno_tab(table_iterator);
    table_iterator := table_iterator - 1;
    return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure **insert_value**(empno employee.empno%type,
    old_empno employee.empno%type,
    new_fname employee.fname%type,
    new_lname employee.lname%type,
    new_bdate employee.bdate%type,
    new_address employee.address%type,
    new_old_ssal employee.salary%type,
    new_ssn employee.ssn%type,
    new_dno employee.dno%type,
    old_sal := old_sal_tab(table_iterator);
    table_iterator := table_iterator - 1;
    return rec;
end insert_value;
new_salary  employee.salary%type,
old_salary  employee.salary%type,
new_ssn   employee.ssn%type,
new_dno   employee.dno%type) is
begin
  table_count := table_count+1;
  emp_tab(table_count) := empno;
  old_emp_tab(table_count) := old_empno;
  fname_tab(table_count) := new_fname;
  lname_tab(table_count) := new_lname;
  bdate_tab(table_count) := new_bdate;
  address_tab(table_count) := new_address;
  sex_tab(table_count) := new_sex;
  salary_tab(table_count) := new_salary;
  old_sal_tab(table_count) := old_salary;
  ssn_tab(table_count) := new_ssn;
  dno_tab(table_count) := new_dno;
end insert_value;

end emp_tab_pkg;
CREATE OR REPLACE PACKAGE SSN_TAB_PKG is
  type rec_type is record(
    new_ssn employee.ssn%type,
    old_ssn employee.ssn%type);
  procedure clear_table;
  procedure set_iterator;
  function more_in_table return boolean;
  function next_rec return rec_type;
  procedure insert_value(new_ssn employee.ssn%type,
    old_ssn employee.ssn%type);
end ssn_tab_pkg;

CREATE OR REPLACE PACKAGE BODY SSN_TAB_PKG is
  type ssn_tab_type is table of employee.ssn%type index by binary_integer;
  new_ssn_tab ssn_tab_type;
  old_ssn_tab ssn_tab_type;
  table_count number;
  table_iterator number := 0;
-- Procedure clear_table: Clear the counter of table to zero.
  procedure clear_table is
    begin
      table_count := 0;
    end clear_table;
-- Procedure set_iterator: Assign the total number of rows
-- after manipulating to the table_iterator variable.
  procedure set_iterator is
    begin
      table_iterator := table_count;
    end set_iterator;
-- Function more_in_table. Check data exists in the table.
  function more_in_table return boolean is
    begin
      return (table_iterator > 0);
    end more_in_table;
-- Function next_rec: Return the data in the table.
  function next_rec return rec_type is rec rec_type;
    begin
      rec.new_ssn := new_ssn_tab(table_iterator);
      rec.old_ssn := old_ssn_tab(table_iterator);
      table_iterator := table_iterator - 1;
      return rec;
    end next_rec;
-- Procedure insert_value: Insert data into the table
  procedure insert_value(new_ssn employee.ssn%type,
    old_ssn employee.ssn%type) is
    begin
      table_count := table_count+1;
      new_ssn_tab(table_count) := new_ssn;
      old_ssn_tab(table_count) := old_ssn;
    end insert_value;
end ssn_tab_pkg;

CREATE OR REPLACE PACKAGE EMP_CONS_PKG is

old_empno employee.empno%type:= ' ';
new_empno employee.empno%type:= ' ';
procedure emp_num(emp_empno employee.empno%type,
rownum1 out number);
procedure ssn_num(emp_ssn employee.ssn%type,
rownum2 out number);
procedure dno_num(emp_dno employee.dno%type,
rownum3 out number);
procedure mgr_num(emp_empno employee.empno%type,
rownum4 out number);
procedure emp_work(emp_empno employee.empno%type,
rownum5 out number);
procedure check_fname(emp_fname employee.fname%type);
procedure check_lname(emp_lname employee.lname%type);
procedure check_bdate(emp_bdate employee.bdate%type);
procedure check_address(emp_address employee.address%type);
procedure check_salary (emp_empno employee.empno%type,
emp_salary employee.salary%type,
emp_ssn employee.ssn%type,
emp_dno employee.dno%type);
procedure enforce_pk_empno_ins(new_empno employee.empno%type);
procedure enforce_pk_empno_upd(rec emp_tab_pkg.rec_type);
procedure emp_upd_cascade(rec emp_tab_pkg.rec_type);
procedure enforce_sex_ins(new_sex employee.sex%type);
procedure enforce_sex_upd(rec emp_tab_pkg.rec_type);
procedure enforce_fk_ssn_ins(new_ssn employee.ssn%type);
procedure enforce_fk_ssn_upd(rec emp_tab_pkg.rec_type);
procedure enforce_fk_dno_ins(new_dno employee.dno%type);
procedure enforce_fk_dno_upd(rec emp_tab_pkg.rec_type);
procedure enforce_sal_upd(rec emp_tab_pkg.rec_type);
procedure emp_del(del_empno employee.empno%type);
procedure audit_ssn(del_ssn employee.ssn%type);
end emp_cons_pkg;

CREATE OR REPLACE PACKAGE BODY EMP_CONS_PKG is

-- Procedure Name: emp_num
-- Purpose: Find the number of employees who have the same employee
-- number as an inserted one.
procedure emp_num(emp_empno employee.empno%type,
rownum1 out number) as
cursor get_emp_row(emp_empno employee.empno%type) is
select count(*) from employee
where empno = emp_empno;
begin
   open get_emp_row(emp_empno);
   fetch get_emp_row into rownum1;
   close get_emp_row;
end emp_num;

-- Procedure Name: ssn_num
-- Purpose: Find the number of employees who have the same employee
-- number as the supervisor number.

procedure ssn_num(emp_ssn employee.ssn%type, rownum2 out number) as
cursor get_ssn_row(emp_ssn employee.ssn%type) is
select count(*) from employee
where emp_ssn = empno;
begin
    open get_ssn_row(emp_ssn);
    fetch get_ssn_row into rownum2;
    close get_ssn_row;
end ssn_num;

-- Procedure Name: dno_num
-- Purpose: Find the number of departments that have the same
-- department number as the inserted one.
procedure dno_num(emp_dno employee.dno%type, rownum3 out number) as
cursor get_dno_row(emp_dno employee.dno%type) is
select count(*) from department
where emp_dno = dno;
begin
    open get_dno_row(emp_dno);
    fetch get_dno_row into rownum3;
    close get_dno_row;
end dno_num;

-- Procedure Name: mgr_num
-- Purpose: Find the number of the department's manager who has the
-- same number as an existing employee.
procedure mgr_num(emp_empno employee.empno%type, rownum4 out number) as
cursor get_mgr_row(emp_empno employee.empno%type) is
select count(*) from department
where emp_empno = mgrssn;
begin
    open get_mgr_row(emp_empno);
    fetch get_mgr_row into rownum4;
    close get_mgr_row;
end mgr_num;

-- Procedure Name: emp_work
-- Purpose: Find the number of employees in EMPLOYEE table
-- which has the same number as an inserting employee number in
-- WORKS_ON table.
procedure emp_work(emp_empno employee.empno%type, rownum5 out number) as
cursor get_work_row(emp_empno employee.empno%type) is
select count(*) from works_on
where emp_empno = empno;
begin
    open get_work_row(emp_empno);
    fetch get_work_row into rownum5;
    close get_work_row;
end emp_work;

-- Procedure Name: check_fname
-- Purpose: Enforce a not null constraint when insert the employee's
-- first name.
procedure **check_fname**(emp_fname employee.fname%type) is
begin
  if emp_fname is null then
    raise_application_error
    (proj_errors.fname_nn, proj_errors.fname_nn_msg);
  end if;
end check_fname;

-- Procedure Name: check_fname_upd
-- Purpose: Enforce a not null constraint when update the employee's first name.
procedure **check_fname_upd**(rec emp_tab_pkg.rec_type) is
begin
  if rec.new_fname is null then
    raise_application_error
    (proj_errors.fname_nn, proj_errors.fname_nn_msg);
  end if;
end check_fname_upd;

-- Procedure Name: check_lname
-- Purpose: Enforce a not null constraint when insert the employee's last name.
procedure **check_lname**(emp_lname employee.lname%type) is
begin
  if emp_lname is null then
    raise_application_error
    (proj_errors.lname_nn, proj_errors.lname_nn_msg);
  end if;
end check_lname;

-- Procedure Name: check_lname_upd
-- Purpose: Enforce a not null constraint when update the employee last name.
procedure **check_lname_upd**(rec emp_tab_pkg.rec_type) is
begin
  if rec.new_lname is null then
    raise_application_error
    (proj_errors.lname_nn, proj_errors.lname_nn_msg);
  end if;
end check_lname_upd;

-- Procedure Name: check_bdate
-- Purpose: Enforce not null and user-defined constraints when insert the employee's birth date.
procedure **check_bdate**(emp_bdate employee.bdate%type) is
begin
  -- Display an error message if the birthdate is null.
  if emp_bdate is null then
    raise_application_error
    (proj_errors.bdate_nn, proj_errors.bdate_nn_msg);
  else
    -- Check user-defined constraint: the age of employees must by between 16 and 65.
    if (((sysdate - emp_bdate)/365) < 18) or
       (((sysdate - emp_bdate)/365) > 65) then
      raise_application_error(proj_errors.age_limit,
-- Procedure Name: check_bdate
-- Purpose: Enforce not null and user-defined constraints when update the employee’s birth date.
Procedure check_bdate_upd (rec emp_tab_pkg.rec_type) is
begin
if rec.new_bdate is null then
raise_application_error(proj_errors.bdate_nn, proj_errors.bdate_nn_msg);
else
if ((sysdate - rec.new_bdate)/365) < 18 or
   ((sysdate - rec.new_bdate)/365) > 65 then
raise_application_error(proj_errors.age_limit, proj_errors.age_limit_msg);
end if;
end if;
end check_bdate_upd;

-- Procedure Name: check_address
-- Purpose: Enforce a not null constraint when insert the employee’s address.
procedure check_address(emp_address employee.address%type) is
begin
if emp_address is null then
raise_application_error(proj_errors.address_nn, proj_errors.address_nn_msg);
end if;
end check_address;

-- Procedure Name: check_address_upd
-- Purpose: Enforce a not null constraint when update the employee’s birth date.
procedure check_address_upd (rec emp_tab_pkg.rec_type) is
begin
if rec.new_address is null then
raise_application_error(proj_errors.address_nn, proj_errors.address_nn_msg);
end if;
end check_address_upd;

-- Procedure Name: check_salary
-- Purpose: Enforce integrity constraints when insert the employee’s salary.
procedure check_salary(emp_empno employee.empno%type,
   emp_salary employee.salary%type,
   emp_ssn employee.ssn%type,
   emp_dno employee.dno%type)
is ssn_sal employee.salary%type;
mgr_sal employee.salary%type;
mgr_ssn department.mgrssn%type;
countt number;
begin
   -- Check a not null constraint.
if emp_salary is null then
    raise_application_error
    (proj_errors.salary_nn, proj_errors.salary_nn_msg);
else
    -- Check a constraint: the salary of employee cannot be negative.
    if emp_salary < 0 then
        raise_application_error
        (proj_errors.salary_ck, proj_errors.salary_ck_msg);
    end if;
end if;

-- Check a user-defined constraint: An employee cannot have a higher salary than the supervisor.
select count(*) into countt
from employee
where emp_ssn = empno;
if countt > 0 then
    select salary into ssn_sal
    from employee
    where emp_ssn = empno;
    if (emp_salary > ssn_sal) then
        raise_application_error(proj_errors.sal_over_ssn,
            proj_errors.sal_over_ssn_msg);
    end if;
end if;

-- Check a user-defined constraint: An employee cannot have a higher salary than the manager of department.
select count(*) into countt
from department
where dno = emp_dno;
if countt > 0 then
    select mgr_ssn into mgr_ssn
    from department
    where dno = emp_dno;
    if mgr_ssn is not null then
        select salary into mgr_sal
        from employee
        where empno = mgr_ssn;
        if (emp_salary > mgr_sal) then
            raise_application_error(proj_errors.sal_over_mgr,
                proj_errors.sal_over_mgr_msg);
        end if;
    end if;
end if;
end check_salary;

-- Procedure Name: enforce_pk_empno_ins
-- Purpose: Enforce a primary key constraint when insert a new employee.
procedure enforce_pk_empno_ins (new_empno employee.empno%type)
as total number;

begin
  -- Display an error message if the new employee number is null.
  if (new_empnd is null) then
    raise_application_error
      (proj_errors.empno_nn, proj_errors.empno_nn_msg);
  else
    -- Display an error message if the new employee already exists in
    the table.
    emp_num(new_empno, total);
    if total > 0 then
      raise_application_error(proj_errors.empno_uk, proj_errors.empno_uk_msg);
    end if;
  end if;
end enforce_pk_empno_ins;

-- Procedure Name: enforce_pk_empno_upd
-- Purpose: Enforce a primary key constraint when update the employee number
procedure enforce_pk_empno_upd(rec emp_tab_pkg.rec_type) is countt number;
begin
  -- Display an error message if the employee number is updated to
  a null value.
  if rec.empno is null then
    raise_application_error
      (proj_errors.empno_nn, proj_errors.empno_nn_msg);
  else
    -- Display error message if the updating empno is not unique.
    select count(*) into countt
      from employee
      where empno = rec.empno;
    if countt > 1 then
      raise_application_error(proj_errors.empno_uk, proj_errors.empno_uk_msg);
    else
      emp_upd_cascade(rec);
    end if;
  end if;
end enforce_pk_empno_upd;

-- Procedure Name: emp_upd_cascade
-- Purpose: Update cascade when update an employee number
procedure emp_upd_cascade(rec emp_tab_pkg.rec_type) is countt number;
begin
  -- If this employee is a manager of department,
  -- update the manager of department to a new one.
  select count(*) into countt
    from department
    where mgrssn = rec.old_empno;
if countt > 0 then
  update department
  set mgrssn = rec.empno
  where mgrssn = rec.old_empno;
end if;
-- If this employee is in the WORKS_ON table,
-- update the employee in the WORKS_ON table to the new employee number.
select count(*) into countt
from works_on
where empno = rec.old_empno;
if countt > 0 then
  update works_on
  set empno = rec.empno
  where empno = rec.old_empno;
end if;
exception
  when no_data_found then
    null;
end emp_upd_cascade;
-- Procedure Name: enforce_sex_ins
-- Purpose: Enforce a check constraint when insert the gender of the employee.
-- Sex can be 'F' female or 'M' male.
procedure enforce_sex_ins(new_sex employee.sex%type) is
begin
  if (upper(new_sex) != 'F') and
     (upper(new_sex) != 'M') then
    raise_application_error
      (proj_errors.sex_ck, proj_errors.sex_ck_msg);
  end if;
end enforce_sex_ins;
-- Procedure Name: enforce_sex_upd
-- Purpose: Enforce a check constraint when update the employee's gender.
procedure enforce_sex_upd(rec emp_tab_pkg.rec_type) is
begin
  if (upper(rec.new_sex) != 'F') and
     (upper(rec.new_sex) != 'M') then
    raise_application_error
      (proj_errors.sex_ck, proj_errors.sex_ck_msg);
  end if;
end enforce_sex_upd;
-- Procedure Name: enforce_fk_ssn
-- Purpose: Enforce a referential integrity constraint when inserting a new supervisor.
-- The supervisor must be an existing employee.
procedure enforce_fk_ssn_ins(new_ssn employee.ssn%type) as total number;
begin
  -- Display error message if the inserting ssn is not
-- an employee in the EMPLOYEE base table.
if new_ssn is not null then
    ssn_num(new_ssn, total);
    if total < 1 then
        raise_application_error
            (proj_errors.ssn_fk, proj_errors.ssn_fk_msg);
    end if;
end if;
end enforce_fk_ssn_ins;

-- Procedure Name: enforce_fk_ssn_upd
-- Purpose: Enforce a referential integrity constraint when updating
-- the supervisor
-- The supervisor must be updated to an existing employee.
procedure enforce_fk_ssn_upd(rec emp_tab_pkg.rec_type)
    is countt number;
begin
    if rec.new_ssn is null then
        return;
    end if;

    select count(*) into countt
    from employee
    where rec.new_ssn = empno;

    if (rec.new_ssn is not null) and (countt < 1) then
        raise_application_error
            (proj_errors.ssn_fk, proj_errors.ssn_fk_msg);
    end if;
end enforce_fk_ssn_upd;

-- Procedure Name: enforce_fk_dno_ins
-- Purpose: Enforce a referential integrity constraint when inserting
-- the department.
-- The department must exist in the DEPARTMENT table.
procedure enforce_fk_dno_ins(new_dno employee.dno%type)
    as total numbers;
begin
    if new_dno is not null then
        dno_num(new_dno, total);
        if total < 1 then
            raise_application_error(proj_errors.dno_fk,
                proj_errors.dno_fk_msg);
        end if;
    end if;
end enforce_fk_dno_ins;

-- Procedure Name: enforce_fk_dno_upd
-- Purpose: Enforce a referential integrity constraint when updating
-- the department.
-- The department must be updated to an existing department.
procedure enforce_fk_dno_upd(rec emp_tab_pkg.rec_type)
    is countt number;
begin
    if rec.new_dno is null then
        return;
    end if;

end if;

select count(*) into countt
from department
where dno = rec.new_dno;

if (rec.new_dno is not null) and (countt < 1) then
    raise_application_error
        (proj_errors.dno_fk, proj_errors.dno_fk_msg);
end if;
end enforce_fk_dno_upd;

-- Procedure Name: enforce_sal_upd
-- Purpose: Enforce user-defined constraint on the salary of employee.
procedure enforce_sal_upd(rec emp_tab_pkg.rec_type)
is sal_avg employee.salary%type;
ssn_sal employee.salary%type;
mgr_sal employee.salary%type;
mgr_ssn department.mgrssn%type;
countt1 number;
begin
    -- Check a not null constraint on the salary of employee.
    if rec.new_salary is null then
        raise_application_error
            (proj_errors.salary_nn, proj_errors.salary_nn_msg);
    else
        -- Check the salary cannot be negative.
        if rec.new_salary < 0 then
            raise_application_error(proj_errors.salary_ck,
                proj_errors.salary_ck_msg);
        end if;
    end if;

    -- Check a user-defined constraint:
    -- An employee older than 60 years old cannot get a raise in salary.
    if ((sysdate-rec.new_bdate)/365) > 60 then
        raise_application_error(proj_errors.sal_age_limit,
            proj_errors.sal_age_limit_msg);
    end if;

    -- Check a user-defined constraint:
    -- An employee, not manager, cannot have a raise in salary more than 15%.
    select count(*) into countt1
    from department
    where rec.empno = mgrssn;
    if (countt1 < 1) and
        ((rec.old_salary * 1.15) < rec.new_salary) then
        raise_application_error
            (proj_errors.sal_higher_limit,
                proj_errors.sal_higher_limit_msg);
    end if;

    -- Check a user-defined constraint:
-- An employee cannot have a higher salary than his/her supervisor.
select count(*) into count1
from employee
where empno = rec.new_ssn;
if count1 > 0 then
select salary into ssn_sal
from employee
where empno = rec.new_ssn;
if rec.new_salary > ssn_sal then
raise_application_error(proj_errors.sal_over_ssn,
proj_errors.sal_over_ssn_msg);
end if;
end if;

-- Check user-defined constraint: an employee cannot have a higher salary the manager.
select count(*) into count1
from department
where dno = rec.new_dno;
if count1 > 0 then
select mgrssn into mgr_ssn
from department
where dno = rec.new_dno;
if mgr_ssn is not null then
select salary into mgr_sal
from employee
where empno = mgr_ssn;
if rec.new_salary > mgr_sal then
raise_application_error(proj_errors.sal_over_mgr,
proj_errors.sal_over_mgr_msg);
end if;
end if;
end if;

insert into emp_sal_aud
values(rec.empno, rec.old_salary, rec.new_salary, sysdate,
to_char(sysdate, 'hh24:mi:ss'));
exception
when no_data_found then
null;
end enforce_sal_upd;

-- Procedure Name: emp_del
-- Purpose: Check a referential integrity constraint when delete an employee.
procedure emp_del(del^empno employee.empno%type) as
cursor manager_cur(del_empno employee.empno%type) is
select dno,dname

203
from department
where mgrssn = del_empno;
total number;
department manager_cur%rowtype;
begin
— If this employee is a manager of department, set the manager of department to null.
mgr_num(del_empno, total);
if total > 0 then
— Keep the information of the deleted manager in the manager_aud table
  open manager_cur(del_empno);
  loop
    fetch manager_cur into dept_inf;
    exit when manager_cur%notfound;
    insert into manager_aud
      values(del_empno, dept_inf.dno, dept_inf.dname, sysdate,
             to_char(sysdate,'hh24:mi:ss'));
  end loop;

  update department
  set mgrssn = null
  where mgrssn = del_empno;
end if;
— If this employee is in WORKS_ON table, delete all work records of this employee.
emp_work(del_empno, total);
if total > 0 then
  delete from works_on
  where del_empno = empno;
end if;

exception
when no_data_found then
  null;
end emp_del;

procedure audit_ssn(del_ssn employee.ssn%type) as
cursor del_ssn_cur (del_ssn employee.ssn%type) is
  select ssn, empno
  from employee
  where ssn = del_ssn;
supervisor del_ssn_cur%rowtype;
begin
  open del_ssn_cur(del_ssn);
  loop
    fetch del_ssn_cur into supervisor;
    exit when del_ssn_cur%notfound;
    insert into supervisor_aud
      values(supervisor.ssn, supervisor.empno, sysdate,
             to_char(sysdate,'hh24:mi:ss'));
  end loop;
end audit_ssn;
end emp_cons_pkg;
CREATE OR REPLACE PACKAGE LOC_TAB_PKG is
    check_field number := 0; -- check which field is updated.
    type rec_type is record(
        new_dno location.dno%type,
        old_dno location.dno%type,
        new_dloc location.dlocation%type,
        old_dloc location.dlocation%type);

    procedure clear_table;
    procedure set_iterator;
    function more_in_table return boolean;
    function next_rec return rec_type;
    procedure insert_value(new_dno location.dno%type,
    old_dno location.dno%type,
    new_dloc location.dlocation%type,
    old_dloc location.dlocation%type);
end loc_tab_pkg;

CREATE OR REPLACE PACKAGE BODY LOC_TAB_PKG is
    type dno_tab_type is table of location.dno%type
        index by binary_integer;
    type dloc_tab_type is table of location.dlocation%type
        index by binary_integer;

    new_dno_tab dno_tab_type;
    old_dno_tab dno_tab_type;
    new_dloc_tab dloc_tab_type;
    old_dloc_tab dloc_tab_type;
    table_count number;
    table_iterator number := 0;

    -- Procedure clear_table: Clear the counter of table to zero.
    procedure clear_table is
        begin
            table_count := 0;
        end clear_table;
        -- Procedure set_iterator: Assign the total number of rows -- after manipulating to the table_iterator variable.
    procedure set_iterator is
        begin
            table_iterator := table_count;
        end set_iterator;

    -- Function more_in_table: Check data exists in the table.
    function more_in_table return boolean is
        begin
            return (table_iterator > 0);
        end more_in_table;

    -- Function next_rec: Return the data in the table.
    function next_rec return rec_type is
        rec rec_type;
        begin
            rec.new_dno := new_dno_tab(table_iterator);
        end next_rec;
rec.old_dno := old_dno_tab(table_iterator);
rec.new_dloc := new_dloc_tab(table_iterator);
rec.old_dloc := old_dloc_tab(table_iterator);
table_iterator := table_iterator - 1;
return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure insert_value(new_dno location.dno%type,
old_dno location.dno%type,
new_dloc location.dlocation%type,
old_dloc location.dlocation%type) is
begin
  table_count := table_count+1;
  new_dno_tab(table_count) := new_dno;
  old_dno_tab(table_count) := old_dno;
  new_dloc_tab(table_count) := new_dloc;
  old_dloc_tab(table_count) := old_dloc;
end insert_value;
end loc_tab_pkg;
CREATE OR REPLACE PACKAGE `LOCCONS_PKG` is
procedure enforce_pk_loc(rec loc_tab_pkg.rec_type); 
end loc_cons_pkg;

CREATE OR REPLACE PACKAGE BODY `LOCCONS_PKG` is
-- Procedure Name: enforce_pk_loc
-- Purpose: Check the primary key constraint on both the department number and location.
procedure enforce_pk_loc(rec loc_tab_pkg.rec_type) 
is countt1 number;
   countt2 number;
begin
   select count(*) into countt1
   from department
   where dno = rec.new_dno;
   
   -- Display error message if the inserting department is null.
   if rec.new_dno is null then
      raise_application_error
      (proj_errors.dno_nn, proj_errors.dno_nn_msg);
   else
      
      -- Display error message if the department does not exist in the DEPARTMENT
      if countt1 < 1 then
         raise_application_error
         (proj_errors.dno_fk, proj_errors.dno_fk_msg);
      end if;
   end if;
   
   -- Display error message if the department location is null.
   if rec.new_dloc is null then
      raise_application_error(loc_errors.dlocation_nn,
         loc_errors.dlocation_nn_msg);
   end if;
   
   -- Check the inserting department number and department location must be unique.
   select count(*) into countt2
   from location
   where dno = rec.new_dno and
         dlocation = rec.new_dloc;
   
   -- Display error if the department and location is not unique.
   if countt2 > 1 then
      raise_application_error(loc_errors.dno_dloc_uk,
         loc_errors.dno_dloc_uk_msg);
   end if;
end enforce_pk_loc;
end loc_cons_pkg;

207
PROJECT:

CREATE OR REPLACE PACKAGE PROJ_TAB_PKG is
  check_field number := 0; -- check which field is updated.
  type rec_type is record(
    new_pnum project.pnumber%type,
    old_pnum project.pnumber%type,
    new_dno project.dno%type,
    new_pname project.pname%type);

  procedure clear_table;
  procedure set_iterator;
  function more_in_table return boolean;
  function next_rec return rec_type;
  procedure insert_value(new_pnum project.pnumber%type,
                         old_pnum project.pnumber%type,
                         new_dno project.dno%type,
                         new_pname project.pname%type);
end proj_tab_pkg;

CREATE OR REPLACE PACKAGE BODY PROJ_TAB_PKG is
  type pnum_tab_type is table of project.pnumber%type
    index by binary_integer;
  type dno_tab_type is table of project.dno%type
    index by binary_integer;
  type pname_tab_type is table of project.pname%type
    index by binary_integer;

  new_pnum_tab pnum_tab_type;
  old_pnum_tab pnum_tab_type;
  new_dno_tab dno_tab_type;
  new_pname_tab pname_tab_type;
  table_count number := 0;
  table_iterator number := 0;

  -- Procedure clear_table: Clear the counter of table to zero.
  procedure clear_table is
    begin
      table_count := 0;
    end clear_table;

  -- Procedure set_iterator: Assign the total number of rows
  -- after manipulating to the table_iterator variable.
  procedure set_iterator is
    begin
      table_iterator := table_count;
    end set_iterator;

  -- Function more_in_table. Check data exists in the table.
  function more_in_table return boolean is
    begin
      return (table_iterator > 0);
    end more_in_table;

  -- Function next_rec: Return the data in the table.
  function next_rec return rec_type is rec rec_type;
begin
  rec.new_pnum := new_pnum_tab(table_iterator);
  rec.old_pnum := old_pnum_tab(table_iterator);
  rec.new_dno := new_dno_tab(table_iterator);
  rec.new_pname := new_pname_tab(table_iterator);
  table_iterator := table_iterator - 1;
  return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure insert_value(new_pnum project.pnumber%type,
                        old_pnum project.pnumber%type,
                        new_dno project.dno%type,
                        new_pname project.pname%type) is
begin
  table_count := table_count+1;
  new_pnum_tab(table_count) := new_pnum;
  old_pnum_tab(table_count) := old_pnum;
  new_dno_tab(table_count) := new_dno;
  new_pname_tab(table_count) := new_pname;
end insert_value;

end proj_tab_pkg;

209
CREATE OR REPLACE PACKAGE PROJ_CONS_PKG is
procedure pnum_num(proj_pnum project.pnumber%type,
rownum1 out number);
procedure dno_num(proj_dno project.dno%type,
rownum2 out number);
procedure pnum_works_on(proj_pnum project.pnumber%type,
rownum3 out number);
procedure pname_num(proj pname project.pname%type,
rownum4 out number);
procedure enforce_pk_pnum_ins(new_pnumber project.pnumber%type);
procedure enforce_pk_pnum_upd(rec proj_tab_pkg.rec_type);
procedure enforce_fk_pname_ins(new_pname project.pname%type);
procedure enforce_fk_pname_upd(rec proj_tab_pkg.rec_type);
procedure project_del(old_pnum project.pnumber%type,
old_pname project.pname%type,
old_dno project.dno%type);
procedure proj_upd_cascade(rec proj_tab_pkg.rec_type);
procedure enforce_fk_dno_ins(new_dno project.dno%type);
procedure enforce_fk_dno_upd(rec proj_tab_pkg.rec_type);
end proj_cons_pkg;

CREATE OR REPLACE PACKAGE BODY PROJ_CONS_PKG is
-- Procedure Name: pnum_num
-- Purpose: Find the number of projects that have the same number as
-- an inserted project.
procedure pnum_num(proj_pnum project.pnumber%type,
rownum1 out number) as
cursor get_pnum_row(proj_pnum project.pnumber%type) is
select count(*)
from project
where pnumber = proj_pnum;
begin
open get_pnum_row(proj_pnum);
fetch get_pnum_row into rownum1;
close get_pnum_row;
end pnum_num;

-- Procedure Name: dno_num
-- Purpose: Find the number of departments that have the same number
-- as an inserted department.
procedure dno_num(proj_dno project.dno%type,
rownum2 out number) as
cursor get_dno_row(proj_dno project.dno%type) is
select count(*)
from department
where dno = proj_dno;
begin
open get_dno_row(proj_dno);
fetch get_dno_row into rownum2;
close get_dno_row;
end dno_num;

-- Procedure Name: pnum_works_on
-- Purpose: Find the number of projects in the WORKS_ON table.
procedure pnum_works_on(proj_pnum project.pnumber%type,
cursor get_pnum_work(proj_pnum project.pnumber%type) is
select count(*)
from works_on
where pnumber = proj_pnum;
begin
open get_pnum_work(proj_pnum);
fetch get_pnum_work into rownum3;
close get_pnum_work;
end pnum_works_on;

-- Procedure Name: pname_num
-- Purpose: Find the number of projects that have the same name as an
inserted one.
procedure pname_num(proj_pname project.pname%type,
rownum4 out number) as
cursor get_pname_row(proj_pname project.pname%type) is
select count(*)
from project
where upper(pname) = upper(proj_pname);
begin
open get_pname_row(proj_pname);
fetch get_pname_row into rownum4;
close get_pname_row;
end pname_num;

-- Procedure Name: enforce_pk_pnum_ins
-- Purpose: Enforce a primary key constraint when insert a new
project.
procedure enforce_pk_pnum_ins(new_pnumber project.pnumber%type)
as total number;
begin
-- Display error message if the project number is null.
if new_pnumber is null then
    raise_application_error(project_errors.pnum_nn,
project_errors.pnum_nn_msg);
else
    -- Display error message if the project already exists.
pnum_num(new_pnumber, total);
    if total > 0 then
        raise_application_error(project_errors.pnum_uk,
project_errors.pnum_uk_msg);
    end if;
end if;
end enforce_pk_pnum_ins;

-- Procedure Name: enforce_pk_pnum_upd
-- Purpose: Enforce a primary key constraint when update the project
number.
procedure enforce_pk_pnum_upd(rec proj_tab_pkg.rec_type)
is countt number;
begin
    select count(*) into countt
    from project
    where pnumber = rec.new_pnum;
end enforce_pk_pnum_upd;
-- Display error message if the project number is updated to a null value.
  if (rec.new_pnum is null) then
    raise_application_error
      (project_errors.pnum_nn, project_errors.pnum_nn_msg);
  else
    -- Display error message if the project number is updated to an existing one.
    if counttt > 1 then
      raise_application_error
        (project_errors.pnum.uk, project_errors.pnum.uk_msg);
    else
      -- Update cascade project number.
      proj_updCascade(rec);
    end if;
  end if;
end enforce_pk_pnum_upd;

-- Procedure Name: enforce_uk_pname_ins
-- Purpose: Enforce not null and unique key constraint when insert a project's name.
procedure enforce_uk_pname_ins(new_pname projec.pname%type) as
  total number;
begin
  -- Display error message if the project name is null.
  if new_pname is null then
    raise_application_error
      (project_errors.pname_nn, project_errors.pname_nn_msg);
  else
    -- Display error message if the project name already exists.
    pname_num(new_pname, total);
    if total > 0 then
      raise_application_error
        (project_errors.pname.uk, project_errors.pname.uk_msg);
    end if;
  end if;
end enforce_uk_pname_ins;

-- Procedure Name: enforce_uk_pname_upd
-- Purpose: Enforce not null and unique key constraint when update the project name.
procedure enforce_uk_pname_upd(rec proj_tab_pkg.rec_type) is
  countt number;
begin
  select count(*) into countt
  from project
  where upper(pname) = upper(rec.new_pname);
  -- Display error message if the project name is updated to a null value.
  if (rec.new_pname is null) then
    raise_application_error
      (project_errors.pname.nn, project_errors.pname.nn_msg);
  else
    -- Display error message if the project name is updated to an
if countt > 1 then
  raise_application_error
  (project_errors.pname uk, project_errors.pname uk_msg);
end if;
end if;
end enforce uk pname upd;

-- Procedure Name: project_del
-- Purpose: Enforce a referential integrity constraint when delete a project.
procedure project_del(old_pnum project.pnumber%type,
  old_pname project.pname%type, old_dno project.dno%type)
  is countt number;
begin
  if countt > 0 then
    delete from works_on
    where pnumber = old_pnum;
  end if;

  -- Keeps the information of deleted project in the project_aud audit table.
  select d.mgrssn into mgr
  from department d
  where d.dno = old_dno;

  select d.dname into dept_dname
  from department d
  where d.dno = old_dno;

  insert into project_aud
    values(old_pnum, old_pname, old_dno, dept_dname, mgr, sysdate,
           to_char(sysdate, 'hh24:mi:ss'));
end project_del;

-- Procedure Name: proj_updCascade
-- Purpose: Update cascade the project in the WORKS_ON table when update a project number.
procedure proj_updCascade(rec proj_tab_pkg.rec_type)
  is countt number;
begin
  -- Update project number in the WORKS_ON table when update a project number.
  select count(*) into countt
  from works_on
  where pnumber = rec.old_pnum;

  if countt > 0 then
    update works_on
    set pnumber = rec.new_pnum
    where pnumber = rec.old_pnum;
  end if;
end proj_updCascade;
update works_on
set pnumber = rec.new_pnum
where pnumber = rec.old_pnum;
end if;
end proj_upd_cascade;

-- Procedure Name: enforce_fk_dno_ins
-- Purpose: Enforce a referential integrity constraint when insert a
-- department
-- A department must exist in the DEPARTMENT table.
procedure enforce_fk_dno_ins(new_dno project.dno%type)
as total number;
begin
-- Check a project must be controlled by a department.
-- Display an error message if the department number is null.
if new_dno is null then
raise_application_error (project_errors.proj_dno_fk,
project_errors.proj_dno_fk_msg);
end if;
-- Display an error message if the department does not exist in
the DEPARTMENT table.
if new_dno is not null then
dno_num (new_dno, total);
if total < 1 then
raise_application_error (project_errors.proj_control_dno,
project_errors.proj_control_dno_msg);
end if;
end if;
end enforce_fk_dno_ins;

-- Procedure Name: enforce_fk_dno_upd
-- Purpose: Enforce a referential integrity constraint when update a
-- department.
procedure enforce_fk_dno_upd (rec proj_tab_pkg.rec_type)
is countt number;
begin
-- Display an error message if the department number is updated
to a null value.
if rec.new_dno is null then
raise_application_error (project_errors.proj_control_dno,
project_errors.proj_control_dno_msg);
end if;
-- Display error message if the department is not updated to an
existing department.
select count(*) into countt
from department
where dno = rec.new_dno;
if (rec.new_dno is not null) and (countt < 1) then
raise_application_error
(project_errors.proj_control_dno,
project_errors.proj_control_dno_msg)
end if;
end enforce_fk_dno_upd;
end proj_cons_pkg;
CREATE OR REPLACE PACKAGE WORKS_ON_TAB_PKG is
    check_field number := 0; -- check which field is updated.
    type rec_type is record(
        new_empno works_on.empno%type,
        old_empno works_on.empno%type,
        new_pnum works_on.pnumber%type,
        old_pnum works_on.pnumber%type);

    procedure clear_table;
    procedure set_iterator;
    function more_in_table return boolean;
    function next_rec return rec_type;
    procedure insert_value(new_empno works_on.empno%type,
                            old_empno works_on.empno%type,
                            new_pnum works_on.pnumber%type,
                            old_pnum works_on.pnumber%type);

end works_on_tab_pkg;

CREATE OR REPLACE PACKAGE BODY WORKS_ON_TAB_PKG is
    type empno_tab_type is table of works_on.empno%type
        index by binary_integer;
    type pnum_tab_type is table of works_on.pnumber%type
        index by binary_integer;
    new_empno_tab empno_tab_type;
    old_empno_tab empno_tab_type;
    new_pnum_tab pnum_tab_type;
    old_pnum_tab pnum_tab_type;
    table_count number;
    table_iterator number := 0;

    -- Procedure clear_table: Clear the counter of table to zero.
    procedure clear_table is
        begin
            table_count := 0;
        end clear_table;

    -- Procedure set_iterator: Assign the total number of rows
    -- after manipulating to the table_iterator variable.
    procedure set_iterator is
        begin
            table_iterator := table_count;
        end set_iterator;

    -- Function more_in_table: Check data exists in the table.
    function more_in_table return boolean is
        begin
            return (table_iterator > 0);
        end more_in_table;

    -- Function next_rec: Return the data in the table.
    function next_rec return rec_type is rec rec_type;
        begin
            rec.new_empno := new_empno_tab(table_iterator);
        end next_rec;

end works_on_tab_pkg;
rec.old_empno := old_empno_tab(table_iterator);
rec.new_pnum := new_pnum_tab(table_iterator);
rec.old_pnum := old_pnum_tab(table_iterator);
table_iterator := table_iterator - 1;
return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure insert_value (new_empno works_on.empno%type,
    old_empno works_on.empno%type,
    new_pnum works_on.pnumber%type,
    old_pnum works_on.pnumber%type) is
begin
    table_count := table_count+1;
    new_empno_tab(table_count) := new_empno;
    old_empno_tab(table_count) := old_empno;
    new_pnum_tab(table_count) := new_pnum;
    old_pnum_tab(table_count) := old_pnum;
end insert_value;

end works_on_tab_pkg;
CREATE OR REPLACE PACKAGE WORKS_ON_CONS_PKG is

procedure empno_num(work_empno works_on.empno%type, rownum1 out number);
procedure pnum_num(work_pnum works_on.pnumber%type, rownum2 out number);
procedure work_num(work_empno works_on.empno%type, work_pnum works_on.pnumber%type, rownum3 out number);
procedure mgrssn_num(new_empno works_on.empno%type, rownum4 out number);
procedure proj_count(new_empno works_on.empno%type, rownum5 out number);
procedure check_hours(new_hours works_on.hours%type);
procedure enforce_fk_empno(new_empno works_on.empno%type);
procedure enforce_fk_pnum(new_pnum works_on.pnumber%type);
procedure enforce_pk_work_ins(new_empno works_on.empno%type, new_pnum works_on.pnumber%type);
procedure enforce_pk_work_upd(rec works_on_tab_pkg.rec_type);
procedure enforce_work_ins(new_empno works_on.empno%type, new_pnum works_on.pnumber%type);
procedure enforce_work_upd(rec works_on_tab_pkg.rec_type);

end works_on_cons_pkg;

CREATE OR REPLACE PACKAGE BODY WORKS_ON_CONS_PKG is

-- Procedure Name: empno_num
-- Purpose: Find the number of employees who have the same empno as an inserted employee.
procedure empno_num(work_empno works_on.empno%type, rownum1 out number) as
cursor get_empno_row(work_empno works_on.empno%type) is
select count(*) from employee
where empno = work_empno;
begin
open get_empno_row(work_empno);
fetch get_empno_row into rownum1;
close get_empno_row;
end empno_num;

-- Procedure Name: pnum_num
-- Purpose: Find the number of project numbers that have the same number as an inserted project.
procedure pnum_num(work_pnum works_on.pnumber%type, rownum2 out number) as
cursor get_pnum_row(work_pnum works_on.pnumber%type) is
select count(*) from project
where pnumber = work_pnum;
begin
open get_pnum_row(work_pnum);
fetch get_pnum_row into rownum2;
close get_pnum_row;
end pnum_num;

-- Procedure Name: work_num
-- Purpose: Find the number of rows in the WORKS_ON table,
-- that has the same empno and pnumber as an inserting one.
procedure work_num(work_empno works_on.empno%type,
                      work_pnum works_on.pnumber%type,
                      rownum3 out number) as
    cursor get_works_row(work_empno works_on.empno%type,
                             work_pnum works_on.pnumber%type) is
        select count(*)
        from works_on
        where (empno = work_empno) and
              (pnumber = work_pnum);
begin
    open get_works_row (work_empno, work_pnum) ;
    fetch get_works_row into rownum3;
    close get_works_row;
end work_num;

-- Procedure Name: mgrssn_num
-- Purpose: Find the number of department's manager
-- who has the same number as an inserted employee.
procedure mgrssn_num(new_empno works_on.empno%type,
                       rownum4 out number) as
    cursor get_mgrssn_num(new_empno works_on.empno%type) is
        select count(*)
        from department
        where mgrssn = new_empno;
    begin
        open get_mgrssn_num (new_empno) ;
        fetch get_mgrssn_num into rownum4;
        close get_mgrssn_num;
    end mgrssn_num;

-- Procedure Name: proj_count
-- Purpose: Find the number of projects that an employee works.
procedure proj_count(new_empno works_on.empno%type,
                      rownum5 out number) as
    cursor get_proj_row(new_empno works_on.empno%type) is
        select count(*)
        from works_on
        where empno = new_empno;
    begin
        open get_proj_row (new_empno) ;
        fetch get_proj_row into rownum5;
        close get_proj_row;
    end proj_count;

-- Procedure Name: check_hours
-- Purpose: Check an employee work hours cannot be negative.
procedure check_hours(new_hours works_on.hours%type) is
begin
    -- Display error message if an employee work hours are negative.
    if new_hours < 0 then
        raise_application_error
        (works_on_errors.hours_ck, works_on_errors.hours_ck_msg);
    end if;
end check_hours;
-- Procedure Name: enforce_fk_empno
-- Purpose: Enforce not null and referential integrity constraints when insert an employee.
-- An employee must exist in the EMPLOYEE table.
procedure enforce_fk_empno(new_empno works_on.empno%type)
as total number;
begin
    -- Display an error message if the employee number is null
    if new_empno is null then
        raise_application_error
        (proj_errors.empno_nn, proj_errors.empno_nn_msg);
    else
        -- Display error if the employee does not exist in the EMPLOYEE table.
        empno_num(new_empno, total);
        if total < 1 then
            raise_application_error
            (works_on_errors.empno_fk, works_on_errors.empno_fk_msg);
        end if;
    end if;
end enforce_fk_empno;

-- Procedure Name: enforce_fk_pnum
-- Purpose: Enforce not null and referential integrity constraints when insert a project.
-- A project must exist in the PROJECT table.
procedure enforce_fk_pnum(new_pnum works_on.pnumber%type)
as total number;
begin
    -- Display error message if the project number is null.
    if new_pnum is null then
        raise_application_error
        (project_errors.pnum_nn, project_errors.pnum_nn_msg);
    else
        -- Display error message if the project does not exist in the PROJECT table.
        pnum_num(new_pnum, total);
        if total < 1 then
            raise_application_error
            (works_on_errors.pnum_fk, works_on_errors.pnum_fk_msg);
        end if;
    end if;
end enforce_fk_pnum;

-- Procedure Name: enforce_pk_work_ins
-- Purpose: Enforce a primary key constraint on both the employee number and project number for an insert operation.
procedure enforce_pk_work_ins(new_empno works_on.empno%type,
                                new_pnum works_on.pnumber%type)
as total number;
begin
    work_num(new_empno, new_pnum, total);
    if total > 0 then

enforce_pk_work_upd:

-- Purpose: Enforce a primary key constrain on both the employee number and project number for an update operation.

procedure enforce_pk_work_upd(rec works_on_tab_pkg.rec_type)

begin

-- Display error message if the employee number is updated to a null value.
if rec.new_empno is null then
  raise_application_error
    (proj_errors.empno_nh, proj_errors.empno_nh_msg);
else
  -- Display error message if this employee is not updated to an existing employee.
  if countt1 < 1 then
    raise_application_error
      (works_on_errors.empno_fk, works_on_errors.empno_fk_msg);
  end if;
end if;

-- Display error message if the updating pnumber is null.
if rec.new_pnum is null then
  raise_application_error
    (project_errors.pnum_nh, project_errors.pnum_nh_msg);
else
  -- Display error message if this project is not updated to an existing project.
  if countt2 < 1 then
    raise_application_error
      (works_on_errors.pnum_fk, works_on_errors.pnum_fk_msg);
  end if;
end if;

-- Check an updating empno and pnumber must be unique.
select count(*) into countt3 from works_on
where (empno = rec.new_empno) and
  (pnumber = rec.new_pnum);

-- Display error message if the updating of empno -- and pnumber is not unique in WORKS_ON table.
if countt1 > 1 then
   raise_application_error(works_on_errors.empno_pnum_uk,
   works_on_errors.empno_pnum_uk_msg);
end if;

enforce_work_upd(rec);
end enforce_pk_work_upd;

-- Procedure Name: enforce_work_ins
-- Purpose: Enforce a user-defined constraint.
-- Employee, not manager, cannot work more than 10 projects.
Procedure enforce_work_ins(new_empno works_on.empno%type,
   new_pnum works_on.pnumber%type)
   is countt1 number;
   countt2 number;
begin
   -- Check the employee is a manager or not.
   mgrssn_num(new_empno, countt1);
   -- Count the number of projects that an employee works.
   proj_count(new_empno, countt2);
   -- Display an error message if the employee, not manager,
   -- works more than 10 projects.
   if (countt1 < 1) and (countt2 >= 10) then
      raise_application_error(works_on_errors.proj_limit,
      works_on_errors.proj_limit_msg);
   end if;
end enforce_work_ins;

-- Procedure Name: enforce_work_upd
-- Purpose: Enforce a user-defined constraint.
-- Employee, not manager, cannot work more than 10 projects.
Procedure enforce_work_upd(rec works_on_tab_pkg.rec_type)
   is countt1 number;
   countt2 number;
begin
   -- Check an employee is a manager or not.
   select count(*) into countt1
   from department
   where mgrssn = rec.new_empno;
   -- Count the number of projects that an employee works.
   select count(*) into countt2
   from works_on
   where empno = rec.new_empno;
   -- Display error message if an employee, not manager, works more
   -- than 10 projects.
   if (countt1 < 1) and (countt2 > 10) then
      raise_application_error(works_on_errors.proj_limit,
      works_on_errors.proj_limit_msg);
   end if;
end enforce_work_upd;
end works_on_cons_pkg;
The COMPANY2 Database Source Code in ORACLE 8

TRIGGERS:

EMPLOYEE:

CREATE OR REPLACE TRIGGER "COMPANY2"."EMP_BIS"
BEFORE INSERT ON "COMPANY2"."EMPLOYEE"
begin
    emp_tab_pkg.clear_table;
end emp_bis;

CREATE OR REPLACE TRIGGER "COMPANY2"."EMP_AIR"
AFTER INSERT ON "COMPANY2"."EMPLOYEE"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    emp_cons_pkg.check_bdate(:new.bdate);
    emp_tab_pkg.insert_value(:new.empno, :new.bdate, :new.ssn, :new.salary, :old.salary, :new.dno);
end emp_air;

CREATE OR REPLACE TRIGGER "COMPANY2"."EMP_AIS"
AFTER INSERT ON "COMPANY2"."EMPLOYEE"
begin
    emp_tab_pkg.set_iterator;
    while emp_tab_pkg.more_in_table loop
        emp_cons_pkg.enforce_sal_ins(emp_tab_pkg.next_rec);
    end loop;
end emp_ais;

CREATE OR REPLACE TRIGGER "COMPANY2"."EMP_BUS"
BEFORE UPDATE ON "COMPANY2"."EMPLOYEE"
begin
    emp_tab_pkg.clear_table;
end emp_bus;

CREATE OR REPLACE TRIGGER "COMPANY2"."EMP_AUR"
AFTER UPDATE OF "BDATE", "SALARY" ON "COMPANY2"."EMPLOYEE"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    if updating ('bdate') then
        emp_cons_pkg.check_bdate(:new.bdate);
    end if;

    if updating ('salary') then
        emp_tab_pkg.insert_value(:new.empno, :new.bdate, :new.ssn, :new.salary, :old.salary, :new.dno);
        emp_tab_pkg.check_field := 3;
    end if;
end emp_aur;
CREATE OR REPLACE TRIGGER "COMPANY2"."EMP_AUS"
AFTER UPDATE ON "COMPANY2"."EMPLOYEE"
begin
    emp_tab_pkg.set_iterator;
    while emp_tab_pkg.more_in_table loop
        if emp_tab_pkg.check_field = 3 then
            emp_cons_pkg.enforce_sal_upd(emp_tab_pkg.next_rec);
        end if;
    end loop;
end emp_aus;

WORKS_ON:

CREATE OR REPLACE TRIGGER "COMPANY2"."WORKS_ON_BIR"
BEFORE INSERT ON "COMPANY2"."WORKS_ON"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    works_on_cons_pkg.enforce_work_ins(:new.empno, :new.pnumber);
end works_on_bir;

CREATE OR REPLACE TRIGGER "COMPANY2"."WORKS_ON_BUS"
BEFORE UPDATE ON "COMPANY2"."WORKS_ON"
begin
    works_on_tab_pkg.clear_table;
end works_on_bus;

CREATE OR REPLACE TRIGGER "COMPANY2"."WORKS_ON_AUR"
AFTER UPDATE OF "EMPNO", "PNUMBER" ON "COMPANY2"."WORKS_ON"
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
begin
    works_on_tab_pkg.insert_value(:new.empno, :old.empno,
                                :new.pnumber, :old.pnumber);
end works_on_aur;

CREATE OR REPLACE TRIGGER "COMPANY2"."WORKS_ON_AUS"
AFTER UPDATE ON "COMPANY2"."WORKS_ON"
begin
    works_on_tab_pkg.set_iterator;
    while works_on_tab_pkg.more_in_table loop
        works_on_cons_pkg.enforce_work_upd
            (works_on_tab_pkg.next_rec);
    end loop;
end works_on_aus;
CREATE OR REPLACE PACKAGE PROJ_ERRORS is

-- Error codes in EMPLOYEE tables (-20150 to -20199)
sal_over_ssn number := -20151;
sal_higher_limit number := -20152;
sal_age_limit number := -20153;
age_limit number := -20154;
ssn_fk number := -20160;
dno_fk number := -20161;
sal_over_mgr number := -20162;

-- Error messages for EMPLOYEE table.
sal_over_ssn_msg varchar2(100) not null :=
    'The salary of employee cannot be higher than that of his/her supervisor.';
sal_higher_limit_msg varchar2(70) not null :=
    'The salary of employee, not manager, cannot increase more than 15%.';
sal_age_limit_msg varchar2(70) not null :=
    'Employee who is more than 60 years old cannot get a raise in salary.';
age_limit_msg varchar2(70) not null :=
    'The age of employee must be between 18 and 65 years old.';
ssn_fk_msg varchar2(70) not null :=
    'Supervisor must be an existing employee or null value.';
dno_fk_msg varchar2(100) not null :=
    'Department number must be a department in DEPARTMENT table or null value.';

-- Error codes in DEPARTMENT table
mgrssri_fk constant number := -20203;
mgrssri_fk_msg varchar2(100) not null :=
    'The manager of department must be an employee in EMPLOYEE table.';
sal_over_mgr_msg varchar2(100) not null :=
    'The salary of employee cannot be higher than that of the manager of department

-- Error code in WORKS_ON table (-20350 to -20399)
proj_limit constant number := -20350;

-- Error message in WORKS_ON table.
proj_limit_msg varchar2(70) not null :=
    'An employee, not manager, cannot work more than 10 projects.';
end proj_errors;

224
PACKAGES:

EMPLOYEE:

CREATE OR REPLACE PACKAGE EMP_TAB_PKG is
check_field number := 0; -- check which field is updated.
type rec_type is record(
    empno employee.empno%type,
    new_bdate employee.bdate%type,
    new_ssn employee.ssn%type,
    new_salary employee.salary%type,
    old_salary employee.salary%type,
    new_dno employee.dno%type);

procedure clear_table;
procedure set_iterator;
function more_in_table return boolean;
function next_rec return rec_type;
procedure insert_value(empno employee.empno%type,
    new_bdate employee.bdate%type,
    new_ssn employee.ssn%type,
    new_salary employee.salary%type,
    old_salary employee.salary%type,
    new_dno employee.dno%type);
end emp_tab_pkg;

CREATE OR REPLACE PACKAGE BODY EMP_TAB_PKG is

emp_tab emp_tab_type;
bdate_tab bdate_tab_type;
ssn_tab ssn_tab_type;
salary_tab salary_tab_type;
old_sal_tab salary_tab_type;
dno_tab dno_tab_type;
table_count number;
table_iterator number := 0;

-- Procedure clear_table: Clear the counter of table to zero.
procedure clear_table is
begin
    table_count := 0;
end clear_table;

-- Procedure set_iterator: Assign the total number of rows
-- after manipulating to the table_iterator violation.
procedure set_iterator is

end emp_tab_pkg;
begin
  table_iterator := table_count;
end set_iterator;

-- Function more in table: Check data exists in the table.
function more_in_table return boolean is
begin
  return (table_iterator > 0);
end more_in_table;

-- Function next_rec: Return the data in the table.
function next_rec return rec_type is rec rec_type;
begin
  rec.empno := emp_tab(table_iterator);
  rec.new_bdate := bdate_tab(table_iterator);
  rec.new_ssn := ssn_tab(table_iterator);
  rec.new_salary := salary_tab(table_iterator);
  rec.old_salary := old_sal_tab(table_iterator);
  rec.new_dno := dno_tab(table_iterator);
  table_iterator := table_iterator - 1;
  return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure insert_value(empno employee.empno%type,
  new_bdate employee.bdate%type,
  new_ssn employee.ssn%type,
  new_salary employee.salary%type,
  old_salary employee.salary%type,
  new_dno employee.dno%type) is
begin
  table_count := table_count+1;
  emp_tab(table_count) := empno;
  bdate_tab(table_count) := new_bdate;
  ssn_tab(table_count) := new_ssn;
  salary_tab(table_count) := new_salary;
  old_sal_tab(table_count) := old_salary;
  dno_tab(table_count) := new_dno;
end insert_value;

end emp_tab_pkg;
CREATE OR REPLACE PACKAGE EMP_CONS_PKG is
procedure check_bdate(emp_bdate employee.bdate%type);   
procedure enforce_sal_ins(rec emp_tab_pkg.rec_type);      
procedure enforce_sal_upd(rec emp_tab_pkg.rec_type);      
end emp_cons_pkg;

CREATE OR REPLACE PACKAGE BODY EMP_CONS_PKG is
-- Procedure Name: check_bdate
-- Purpose: Enforce a not null and user-defined constraints
-- when insert the birth date of employee.
procedure check_bdate(emp_bdate employee.bdate%type) is

begin
-- Display an error message if the age of employees is not
between 18 and 65
  if (((sysdate - emp_bdate)/365) < 18) or
    ((sysdate - emp_bdate)/365) > 65) then
    raise_application_error(proj_errors.age_limit,
      proj_errors.age_limit_msg);
  end if;
end check_bdate;

-- Procedure Name: enforce_sal_ins
-- Purpose: Enforce user-defined constraints when insert the salary of
employee.
procedure enforce_sal_ins(rec emp_tab_pkg.rec_type) is
  mgr_sal employee.salary%type;
  ssn_sal employee.salary%type;
  mgr_ssn department.mgrssn%type;
  counttl number;

begin
-- Check constraint: An employee cannot have a higher
-- salary than his/her supervisor.
-- Find the supervisor's salary.
  select count(*) into counttl from employee
  where empno = rec.new_ssn;

  if (counttl > 0) then
    select salary into ssn_sal from employee
    where empno = rec.new_ssn;

    if rec.new_salary > ssn_sal then
      raise_application_error
        (proj_errors.sal_over_ssn,
          proj_errors.sal_over_ssn_msg);
    end if;
  end if;

-- Check user-defined constraint:
-- An employee cannot have a higher salary than the manager of
department.
  select count(*) into counttl from department
  where dno = rec.new_dno;

end enforce_sal_ins;

end emp_cons_pkg;

227
if (counttl > 0) then
    select mgrssn into ingr__ssn
    from department
    where dno = rec.new_dno;

    if mgr_ssn is not null then
        select salary into mgr_sal
        from employee
        where empno = mgr_ssn;

        if rec.new_salary > mgr_sal then
            raise_application_error
                (proj_errors.sal_over_mgr,
                proj_errors.sal_over_mgr_msg);
        end if;
    end if;
end if;
end enforce_sal_ins;

-- Procedure Name: enforce_sal_upd
-- Purpose: Enforce user-defined constraints when update the salary of employee.
procedure enforce_sal_upd (rec emp_tab_pkg.rec_type)
is mgr_sal employee.salary%type;
    ssn_sal employee.salary%type;
    mgr_ssn department.mgrssn%type;
    counttl number;
begin
    -- Check constraint: An employee who is older than 60 years old
    -- cannot get a raise in salary.
    if ((sysdate-rec.new_bdate)/365) > 60 then
        raise_application_error(proj_errors.sal_age_limit,
                               proj_errors.sal_age_limit_msg);
    end if;
    -- Check constraint: An employee, who is not a manager of department,
    -- cannot have a raise in salary more than 15%.
    select count(*) into counttl
    from department
    where rec.empno = mgrssn;

    if (counttl < 1) and
        ((rec.old_salary * 1.15) < rec.new_salary) then
        raise_application_error
            (proj_errors.sal_higher_limit,
            proj_errors.sal_higher_limit_msg);
    end if;
    -- Check constraint: An employee cannot have a higher
    -- salary than his/her supervisor.
    -- Find the supervisor's salary.
    select count(*) into counttl
    from employee
    where empno = rec.new_ssn;

    if (counttl > 0) then
select salary into ssn_sal
from employee
where empno = rec.new_ssn;

if rec.new_salary > ssn_sal then
    raise_application_error
        (proj_errors.sal_over_ssn,
         proj_errors.sal_over_ssn_msg);
endif;
endif;

-- Check user-defined constraint:
-- An employee cannot have a higher salary than the manager of
-- department.
select count(*) into countt1
from department
where dno = rec.new_dno;

if (countt1 > 0) then
    select mgrssn into mgr_ssn
    from department
    where dno = rec.new_dno;
    if mgr_ssn is not null then
        select salary into mgr_sal
        from employee
        where empno = mgr_ssn;
        if rec.new_salary > mgr_sal then
            raise_application_error
                (proj_errors.sal_over_mgr,
                 proj_errors.sal_over_mgr_msg);
        endif;
    endif;
endif;

insert into emp_sal_aud
values (rec.empno, rec.old_salary, rec.new_salary, sysdate,
to_char(sysdate,'hh24:mi:ss'));
end enforce_salupd;
end emp_cons_pkg;
WORKS_On:

CREATE OR REPLACE PACKAGE WORKS_ON_TAB_PKG is
  check_field number := 0; -- check which field is updated.
  type rec_type is record(
    new_empno works_on.empno%type,
    old_empno works_on.empno%type,
    new_pnum works_on.pnumber%type,
    old_pnum works_on.pnumber%type);

  procedure clear_table;
  procedure set_iterator;
  function more_in_table return boolean;
  function next_rec return rec_type;
  procedure insert_value(new_empno works_on.empno%type,
                           old_empno works_on.empno%type,
                           new_pnum works_on.pnumber%type,
                           old_pnum works_on.pnumber%type);

end works_on_tab_pkg;

CREATE OR REPLACE PACKAGE BODY WORKS_ON_TAB_PKG is
  type empno_tab_type is table of works_on.empno%type
                   index by binary_integer;
  type pnum_tab_type is table of works_on.pnumber%type
                   index by binary_integer;

  new_empno_tab empno_tab_type;
  old_empno_tab empno_tab_type;
  new_pnum_tab pnum_tab_type;
  old_pnum_tab pnum_tab_type;
  table_count number;
  table_iterator number := 0;

  -- Procedure clear_table: Clear the counter of table to zero.
  procedure clear_table is
    begin
      table_count := 0;
      end clear_table;
  end;

  -- Procedure set_iterator: Assign the total number of rows
  -- after manipulating to the table_iterator variable.
  procedure set_iterator is
    begin
      table_iterator := table_count;
      end set_iterator;
  end;

  -- Function more_in_table: Check data exists in the table.
  function more_in_table return boolean is
    begin
      return (table_iterator > 0);
      end more_in_table;
  end;

  -- Function next_rec: Return the data in the table.
  function next_rec return rec_type is rec rec_type;
    begin
      rec.new_empno := new_empno_tab(table_iterator);
    end
end;
rec.old_empno := old_empno_tab(table_iterator);
rec.new_pnum := new_pnum_tab(table_iterator);
rec.old_pnum := old_pnum_tab(table_iterator);
table_iterator := table_iterator - 1;
return rec;
end next_rec;

-- Procedure insert_value: Insert data into the table
procedure insert_value (new_empno works_on.empno%type,
old_empno works_on.empno%type,
new_pnum works_on.pnumber%type,
old_pnum works_on.pnumber%type) is
begin
  table_count := table_count+1;
  new_empno_tab(table_count) := new_empno;
  old_empno_tab(table_count) := old_empno;
  new_pnum_tab(table_count) := new_pnum;
  old_pnum_tab(table_count) := old_pnum;
end insert_value;

end works_on_tab_pkg;
CREATE OR REPLACE PACKAGE WORKS_ON_CONS_PKG is

procedure mgrssn_num (new_empno works_on.empno%type,
rownum4 out number);  
procedure proj_count(new_empno works_on.empno%type,
rownum5 out number);  
procedure enforce_work_ins(new_empno works_on.empno%type,
new_pnum works_on.pnumber%type);  
procedure enforce_work upd(rec works_on_tab_pkg.rec_type);  
end works_on_cons_pkg;

CREATE OR REPLACE PACKAGE BODY WORKS_ON_CONS_PKG is

Procedure Name: mgrssn_num
-- Purpose: Count the number of employee who
-- has the same number as an inserting empno.
procedure mgrssn_num(new_empno works_on.empno%type,
rownum4 out number) as

cursor get_mgrssn_num(new_empno, works_on.empno%type) is
select count(*)
from department
where mgrssn = new_empno;
begin
open get_mgrssn_num(new_empno);
fetch get_mgrssn_num into rownum4;
close get_mgrssn_num;
end mgrssn_num;

Procedure Name: proj_count
-- Purpose: Count the number of projects in WORKS_ON table
-- that has the same employee number as an updated employee.
Procedure proj_count(new_empno works_on.empno%type,
rownum5 out number) as

cursor get_proj_row(new_empno works_on.empno%type) is
select count(*)
from works_on
where empno = new_empno;
begin
open get_proj_row(new_empno);
fetch get_proj_row into rownum5;
close get_proj_row;
end proj_count;

Procedure Name: enforce_work_ins
-- Purpose: Check user-defined constraint
-- when inserting an employee work record in WORKS_ON table.
-- An employee, not manager, cannot work more than 10 projects.
Procedure enforce_work_ins(new_empno works_on.empno%type,
new_pnum works_on.pnumber%type)
is counttl number;
countt2 number;
begin
-- Check the employee is a manager or not.
mgrssn_num(new_empno, countt1);

-- Count the number of projects which an updating
-- employee works on those projects.
proj_count(new_empno, countt2);

-- Display error if the employee, who is not a department’s manager, works more than 10 projects.
if (countt1 < 1) and (countt2 >= 10) then
  raise_application_error(proj_errors.proj_limit,
    proj_errors.proj_limit_msg);
end if;
end enforce_work_ins;

-- Procedure Name: enforce_work_upd
-- Purpose: Check a user-defined constraint
-- when updating an employee work record in WORKS_ON table.
-- An employee, not manager, cannot work more than 10 projects.
Procedure enforce_work_upd(rec works_on_tab_pkg.rec_type)
  is countt1 number;
  countt2 number;
begin
  -- Find the employee is a manager or not
  -- by counting the number of manager, who has
  -- the same empno as an updating empno.
  select count(*) into countt1
  from department
  where mgrssn = rec.new_empno;

  -- Count the number of projects, which the
  -- employee works on them.
  select count(*) into countt2
  from works_on
  where empno = rec.new_empno;

  -- Display error message if an updating employee,
  -- not a manager, works more than 10 projects.
  if (countt1 < 1) and (countt2 > 10) then
    raise_application_error(proj_errors.proj_limit,
      proj_errors.proj_limit_msg);
  end if;
end enforce_work_upd;
end works_on_cons_pkg;
APPENDIX D: USER MANUAL

This user manual describes the use of SQL SERVER 7.0 and ORACLE 8 to operate the COMPANY database in this project. Examples of manipulating data in the database are also given in this section.

SQL SERVER 7.0 User Manual in the COMPANY Database

To manipulate data in the COMPANY database in SQL SERVER 7.0:

1. Login to the SQL SERVER QUERY ANALYZER by connecting to the SQL SERVER 7.0. This project uses the ZEBRASERVER as a server. (Figure D-1).

   Figure D-1 Login to the SQL SERVER

   ![Login to SQL Server](image)

2. Connect to the database by select the database (COMPANY1 or COMPANY2) from the DB combo box as shown in Figure D-2.
3. Issue a SQL DML statement and execute the operation by clicking the green button (also shown in Figure D-2).

A DELETE, INSERT or UPDATE statement will be manipulated completely if no violation occurs. In case a violation happens, an error message will be displayed differently between the COMPANY1 and COMPANY2 as shown below. The top screen shows an error message made by triggers and the bottom screen gives a system error message. Figure D-3 to Figure D-9 illustrate integrity constraint violations in both COMPANY1 and COMPANY2 databases in SQL SERVER 7.0.
Figure D-3 A Primary Key Violation in the EMPLOYEE Table

```sql
insert into employee

Server: Msg 50106, Level 15, State 1, Line 0
Employee number must be unique.

insert into employee

Server: Msg 50107, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'empl_empno_pk'. Cannot insert duplicate key in object 'empl'.
The statement has been terminated.
```
Figure D-4 A Check Constraint Violation on the Sex Attribute in the EMPLOYEE Table
Figure D-5 A Foreign Key Violation in the EMPLOYEE Table

```
insert into employee
```

Server: Mag 5011, Level 15, State 1, Line 0
Department number must be a department in DEPARTMENT table or null value.

```
insert into employee
```

Server: Mag 547, Level 16, State 1, Line 1
INSERT statement conflicted with COLUMN FOREIGN KEY constraint 'emp_dno_fk'. The conflict occurred: The statement has been terminated.
A salary of an employee cannot be higher than his/her supervisor's salary.
Since SQL SERVER 7.0 cannot apply an update or delete cascade by using the declarative method, the following shows an implementation of triggers to update and delete cascade on the EMPLOYEE table.

Figure D-7 shows before issuing an update or delete operation on the employee number (empno = '333444555').

Figure D-7 Before Update or Delete an Employee in the EMPLOYEE Table
Figure D-8 After an Update Cascade on the EMPLOYEE Table

After updating an employee in the EMPLOYEE table, all data that refer to the updated employee are also updated to the new value ('333333333')--a ssn in the EMPLOYEE table, a mgrssn in the DEPARTMENT table, and the employ in the WORKS_ON table.
After deleting an employee, the employee work records in the WORKS_ON table are also deleted. The ssn in the EMPLOYEE table and the mgrssn in the DEPARTMENT table that refer to the deleted employ will be set to null.

In SQL SERVER 7.0, triggers can also be enabled or disable by performing the following command:

```
ALTER TABLE table_name [ENABLE| DISABLE] TRIGGER trigger_name;
```
In ORACLE 8, users can login to the ORACLE SQL Worksheet or ORACLE SQL*Plus in order to insert, update or delete the data in the COMPANY1 and COMPANY2 databases. Like SQL SERVER 7.0, the COMPANY1 database is implemented by triggers and the COMPANY2 database uses declarative method. Examples an implementation in ORACLE 8 are also given.

To manipulate data in the COMPANY database in ORACLE 8 by using the ORACLE SQL Worksheet:

1. Login to the database requires the user name, password, service name and the status of connection as shown in Figure D-10.

   Figure D-10 Login to the ORACLE SQL Worksheet

```
Username: company1
Password: ********
Service: project
Connect As: Normal
```

2. Issue the SQL DML statements in the bottom part and execute the command. The result of the SQL command will be displayed in the top part as shown in Figure D-11.
Figure D-11 Manipulating Data Using ORACLE SQL Worksheet

```
Oracle SQL Worksheet - project - [Untitled]

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>111222333</td>
</tr>
<tr>
<td>5</td>
<td>Finance</td>
<td>999999999</td>
</tr>
<tr>
<td>3</td>
<td>Headquarters</td>
<td>555666777</td>
</tr>
<tr>
<td>4</td>
<td>Environment</td>
<td>777888999</td>
</tr>
<tr>
<td>10</td>
<td>Temporary</td>
<td>999999999</td>
</tr>
</tbody>
</table>
5 rows selected.
```

```
select *
from department;
```

execute button
To manipulate data in the COMPANY database in ORACLE 8 by using the ORACLE SQL*Plus:

1. login to the database also requires the user name, password and host string (the same as the service name when connect to ORACLE SQL Worksheet).

Figure D-12 Login to the ORACLE SQL*Plus

2. Issue the SQL statement and execute the command (Figure D-13)

Figure D-13 Manipulating Data Using ORACLE SQL*Plus
The following examples use the ORACLE SQL Worksheet tool to manipulate the data in the COMPANY database in ORACLE 8.

Figure D-14 A Primary Key Violation in the DEPARTMENT Table
Figure D-15 A Foreign Key Violation in the DEPARTMENT Table
Figure D-16, D-17 and D-18 illustrate an update or delete cascade in the COMPANY1 database implemented by using triggers.

Figure D-16 Before Update or Delete a Department in the DEPARTMENT Table

All related data that refer to an update or delete department number (dno = 3) is shown above. An updating dno in the DEPARTMENT table will affect the dno in the EMPLOYEE, LOCATION, and PROJECT tables.
After updating a department number (dno = 3 to dno = 6), all related data in the EMPLOYEE, LOCATION, and PROJECT tables are also updated to a new department (dno = 6) as shown in the above figure.
After deleting a department number (dno = 6), the department’s locations in the LOCATION table are also deleted. The department number in the PROJECT tables will be set to a default department (dno=10). The department number in the EMPLOYEE table will be set to null.
Using the declarative method in ORACLE 8 can perform a deleted cascade (See Figure D-20), but an update in the primary key that has other child tables refer to the updated primary key is not permitted as seen in the following figure.

Figure D-19. Disallow an Update Cascade in the Declarative Method

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>DNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456783</td>
<td>3</td>
</tr>
<tr>
<td>123456780</td>
<td>3</td>
</tr>
<tr>
<td>555666777</td>
<td>3</td>
</tr>
<tr>
<td>3 rows selected.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DNO</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Houston</td>
</tr>
<tr>
<td>3</td>
<td>Stafford</td>
</tr>
<tr>
<td>2 rows selected.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>DNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>ReorganizationP2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>ReorganizationP1</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>ReorganizationP3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>ReorganizationP4</td>
<td>3</td>
</tr>
<tr>
<td>4 rows selected.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ORA-02292: integrity constraint (COMPANY2.EMP_DNO_FK) violated - child record found
Figure D-20. A Delete Cascade in the Declarative Method

After deleting a department number, all data in other tables that related to the deleted data will be deleted.
Triggers in ORACLE 8 can be enabled or disable by the following command:

```
ALTER TRIGGER trigger_name [ENABLE| DISABLE];
```
REFERENCES


