Module 1

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 1 Study Guide and Deliverables

Readings:

Required Reding:

- CB6 chapter 15

Reccomended Reading:

- CB6 chapters 6, 7, and 8 - SQL Programming
- Loney chapters 32, 34 and 35 - Oracle Specific

Assignments:

- Assignment 1.0, due Wednesday, July 15 at 5:00 PM ET
- Programming Part 1, due Sunday, July 19 at 6:00 AM ET

Assessments:

- Crediting Sources Tutorial Self-Assessment due Wednesday, July 15 at 5:00 PM ET
- Quiz 1 due Wednesday, July 15 at 5:00 PM ET

Term Project Note:

Term Project Update #1, is due Sunday, July 19 at 6:00 AM ET.

During this first module you should begin to think about what you will be doing for your term project and discuss your ideas with your instructor. Your term project can be

	based on any advanced database topic, including but not limited to XML and databases, database performance measurement or tuning, advanced non-relational databases, decision support databases, data mining, distributed databases, object-oriented databases, object-relational databases, tiered databases, very large databases, or advanced database architectures.
Live Classrooms:	Supplementary Live Session, Tuesday, July 7, 8:00 PM - 10:00 PM ET Current week's assignment review and examples, Wednesday, July 8, 8:00 PM - 9:00 PM ET Live office help, Saturday, July 11, 11:00 AM - 12:00 PM ET Live office help, Monday, July 13, 8:00 PM - 8:45 PM ET

Lecture 1A - Advanced Normalization

Introduction

In this lecture you will continue the study of normalization that you began in MET CS 669, and will learn about denormalization. We will introduce Boyce-Codd normal form and fourth normal form. We will discuss over normalization, which sometimes happens when we apply the normalization rules more thoroughly than is appropriate, resulting in more tables than are useful. We will then discuss denormalization, which reassembles the objects that normalization has disassembled, and does other things to improve performance. In this lesson you will also learn to use object-oriented ways of thinking about the design of relational databases. As you learn to think this way, you will find that you are able to design good, simple, and efficient schemas fairly quickly, without worrying about higher normal form problems.

Learning Outcomes

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Describe Boyce-Codd and Fourth Normal Form
- Determine whether a table is in Boyce-Codd Normal Form
- Renormalize to Boyce-Codd Normal Form
- Determine whether a table is in Fourth Normal Form
- Renormalize to Fourth Normal Form
- Determine when denormalization may improve performance
- Safely denormalize while minimizing the risks of anomalies and inconsistent data
- Design databases using Enhanced Entity-Relationship Modeling
- Design and implement database triggers of medium complexity
- Design and implement database processes and functions

Third and Boyce-Codd Normal Forms

A discussion of third and fourth normal forms is complicated by the fact that there are several popular oversimplified definitions of third normal form and several equivalent correct definitions of third normal form. One oversimplified definition of third normal form is that a table is in third normal form if it is in second normal form and it contains no transitive dependencies. Another oversimplified definition is that a table is in third normal form if every non-key attribute provides information about the key and about nothing else. This works fine if the table has only one determinant, which is a candidate key. Sometimes it happens that a table has more than one determinant. Consider the following example.

> A company database has a COMPUTER table in which they record data for all of their computers. In this company all computers have an asset tag with an ASSET_ID, which uniquely identifies each computer. All computers in this company also have a MAKE, MODEL, and SERIAL_NUMBER, which also uniquely identify each computer. Thus each computer has two candidate keys, the ASSET_ID, and the triple {MAKE, MODEL, and SERIAL_NUMBER}. If the COMPUTER table has columns ASSET_ID, MAKE, MODEL, and SERIAL_NUMBER, then the COMPUTER table has multiple candidate keys.

This is common. For example, many tables representing people have multiple candidate keys.

There are transitive dependencies between the ASSET_ID and the triple {MAKE, MODEL, and SERIAL_NUMBER}. No matter which of the keys that we choose as the primary key, there will still be transitive dependencies between the other candidate key and all non-key columns. Thus this table is not in third normal form (3NF) by these oversimplified definitions of 3NF. Yet there seems to be nothing wrong with this table. It just has two candidate keys, and each of these determines all of the other columns. There is nothing wrong with a table that has more than one candidate key; this occurs frequently. The problem is with the oversimplified definitions of third normal form.

This problem with the oversimplified definitions of third normal form is the reason that our text (e.g. CB6 page 427) and other rigorous sources provide more complex definitions. Third normal form was originally defined by E.F. Codd in 1971 [Codd71], and his definition is general, because it doesn't require keys. Codd's definition states that a relation R is in 3NF if and only if it satisfies both of the following conditions:

- The relation R is in second normal form (2NF), and

- Every non-prime attribute of R is non-transitively dependent on every key of R.

A *non-prime attribute* is an attribute that does not belong to any candidate key of R, and a *transitive dependency* is a functional dependency in which $X \rightarrow Z$ (X determines Z) indirectly, by virtue of $X \rightarrow Y$ and $Y \rightarrow Z$ (where it is not the case that $Y \rightarrow X$).

Our Connolly and Begg text provides us with an equivalent definition of the modern general third normal form, which also does not require that a relation have keys:

> **The general definition of third normal form (3NF)** is a relation that is in 1NF and 2NF in which no non-candidate-key attribute is transitively dependent on any candidate key. In this definition a candidate key attribute is part of any candidate key.

BCNF is not just an improved version of third normal form. BCNF is more restrictive than 3NF. Every relation that is in BCNF is also in 3NF, but not every relation that is in 3NF is in BCNF. There is a good discussion and example of this in CB6 pages 438 and 439.

It can happen that information is lost when a table is normalized from third normal form to BCNF. This is uncommon, so most databases are normalized to BCNF, but you need to be aware that it can occur. There is a good discussion of this in CB6 pages 439 and 440. The key point is that there is a tradeoff between representing all of the functional dependencies and data redundancy. 3NF may represent important functional dependencies that are lost when the relation is normalized to BCNF. If you encounter this situation you have the choice of keeping the relation in 3NF and dealing with the data redundancies or normalizing to BCNF and accepting the loss of the functional dependencies. If your database architecture provides good control of the consistency of redundant data (such as restricting table access to stored procedures that maintain the consistency of redundant data) and the functional dependencies that would be lost are important to represent then it may be best to keep the relation in 3NF.

Summary of the modern versions of 1NF, 2NF, and 3NF

As people developed and used relational databases they noticed that many tables, such as history and audit tables, often had no natural primary key, and still functioned quite well. Over time people developed newer definitions of 1NF, 2NF, and 3NF which are based on functional dependencies and do not require a primary key or even a key. Connolly and Begg present these as general definitions of the normal forms, for example on CB6 page 427. The following definitions are adapted from those in CB6:

> A table is in general first normal form if and only if the data are arranged in tabular form with the data in each cell being atomic, the columns represent the same attributes of entity instances, and all of the attributes of each entity instance are represented in the same row.

Notice that this definition supports the key goals of traditional 1NF without requiring a primary key or even a candidate key.

In the following definitions of the general forms of 2NF and 3NF a *candidate key attribute* is an attribute that is part of any candidate key and a *non-candidate-key attribute* is an attribute that is not part of any candidate key.

> A table is in general second normal form if and only if it is in first normal form and every non-candidate-key attribute is fully functionally dependent on every candidate key. An attribute is fully functionally dependent on another set of attributes if it is functionally dependent on the full set of attributes, but not on any proper subset of the attributes.

This is the generalization of the traditional 2NF which does not require a primary key. Note that all tables in 1NF without candidate keys satisfy this definition of 2NF. Next we consider the modern version of 3NF:

> A table is in general third normal form if and only if it is in second normal form and no non-candidate-key attributes are transitively dependent on any candidate key.

Transitive dependencies occur when a set of columns determines a second set of columns and those columns in turn determine a third set of columns, all in the same table. (If you don't feel that you understand this you can consult CB6 Page 412, which includes an extensive discussion and examples.) Note that this is a generalization of the traditional 3NF definition to the cases where there is no primary key. This definition does not require candidate keys either. We now state the condition for BCNF:

> A table is in Boyce-Codd Normal form if and only if all determinants are super keys.
>
> With the definition of a determinant as a minimal set of attributes that fully functionally determines another set of attributes there is an equivalent definition that a relation is in BCNF if and only if all determinants are candidate keys.

Note that this definition of BCNF does not require that there be a primary key or candidate keys or even that the table have functional dependencies.

In 1982 Carlo Zanio published a 3NF definition that clarifies the relationship between 3NF and BCNF [Zanio82]. His 3NF definition is equivalent to Codd's, but he expressed it differently. This definition states that a table is in 3NF if and only if, for each of its functional dependencies $X \rightarrow Y$, at least one of the following conditions holds:

1. X contains Y ($X \rightarrow Y$ is a trivial functional dependency), or
2. X is a superkey, or
3. Every element of Y-X, the set difference between Y and X, is a *prime attribute.*

The first two conditions are the same as for BCNF. Zaniolo's definition of 3NF also allows the third condition. Thus relations where at least one functional dependency satisfies condition 3, but not condition 1 or 2, is in 3NF but not in BCNF.

References:

[Codd71] Codd, E.F. "Further Normalization of the Data Base Relational Model." (Presented at Courant Computer Science Symposia Series 6, "Data Base Systems," New York City, May 24th–25th, 1971.) IBM

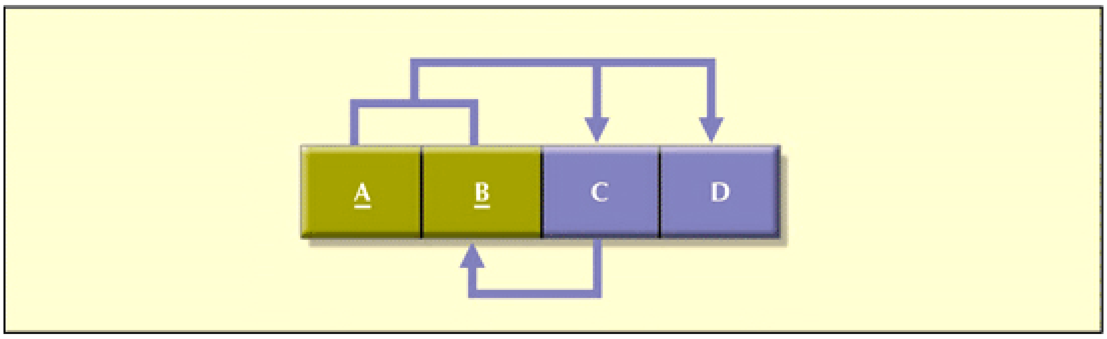
Research Report RJ909 (August 31st, 1971). Republished in Randall J. Rustin (ed.), *Data Base Systems: Courant Computer Science Symposia Series 6*. Prentice-Hall, 1972.

[Zaniolo82] Zaniolo, Carlo. "A New Normal Form for the Design of Relational Database Schemata." *ACM Transactions on Database Systems* 7(3), September 1982.

More on Boyce-Codd Normal Form

A table can be in third normal form but not Boyce-Codd normal form (BCNF). Consider the following functional dependency diagram from the CS 669 text.

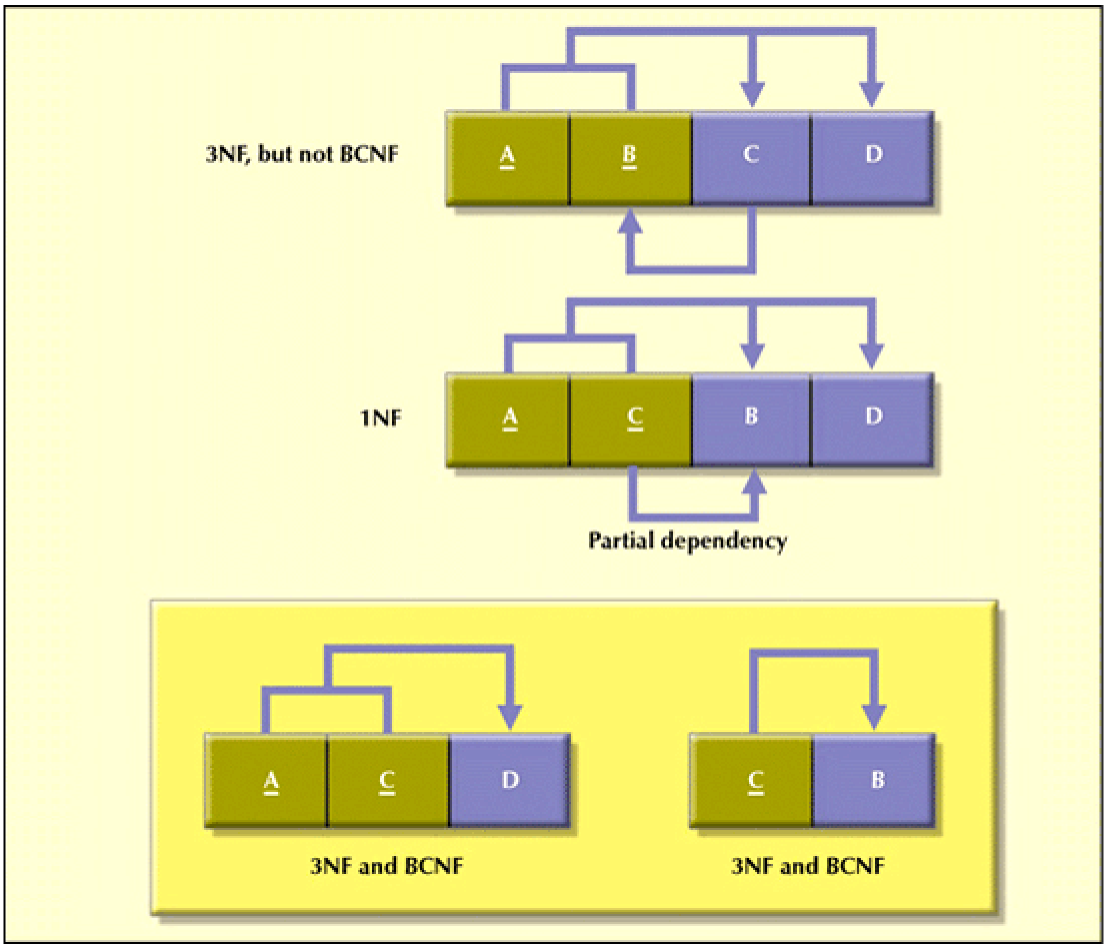
A TABLE THAT IS IN 3NF BUT NOT IN BCNF



This table is in third normal form, because it is in second normal form and there are no transitive dependencies. (That B depends on C is not transitive, because C depends upon both A and B.) Yet the table is not in BCNF, because even though C determines the key column B, C is not a candidate key by itself.

How do we fix this? Because C determines B, we can replace B in the primary key by C. This gives us a table that is not even in second normal form, because C, which is now only a part of the primary key, determines B, which is now a non-key column. Thus B does not depend on the whole key, which violates the second normal form. This is illustrated in the center functional dependency diagram in the following figure:

Decomposition to BCNF



We resolve the problem in the standard way by normalizing this table into two tables, which are both in 3NF and BCNF, as illustrated at the bottom of the figure. An example follows.

Satisfying Both 3NF and BCNF

An example will help clarify this. Consider the following sample data from the text.

Sample Data for a BCNF Conversion

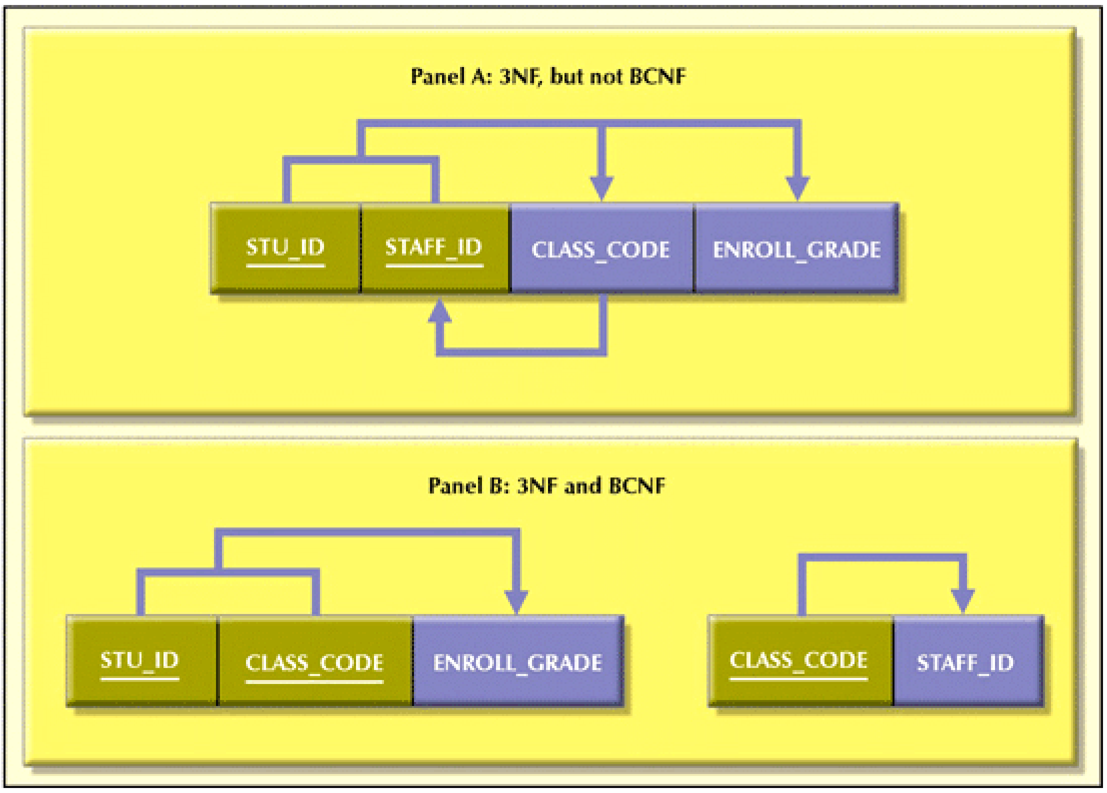
STU_ID	STAFF_ID	CLASS_CODE	ENROLL_GRADE
125	25	21334	A
125	20	32456	C
135	20	28458	B
144	25	27563	C
144	20	32456	B

Consider the functional dependencies.

- CLASS_CODE is really a section code, so CLASS_CODE determines the STAFF_ID of the instructor.
- STAFF_ID does not determine CLASS_CODE, because an instructor may teach more than one section.
- A student can only take one section from an instructor, so the combination of STU_ID and STAFF_ID determines CLASS_CODE and ENROLL_GRADE.

The resulting functional dependency graph looks like Panel A in the following diagram.

Another BCNF Decomposition



This database table is trying to represent two different things:

- An instructor teaching a class
- A student taking a class

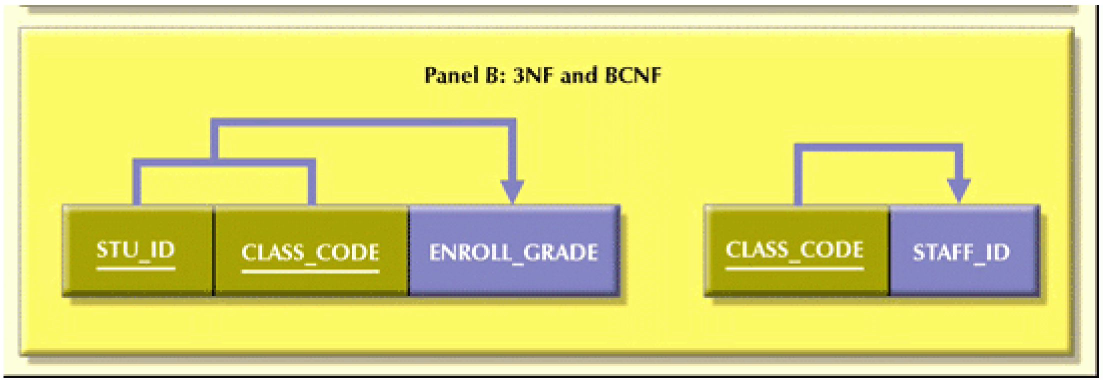
These are quite distinct, with different lifetimes. This causes serious problems, including:

- There is no way to record that an instructor is assigned to a class without also assigning a student to the class at the same time.
- If you delete all of the students from a class you lose the information about who is teaching the class.
- If you change the instructor for a class you have to update all of the rows for the class. While an appropriate update clause can handle this, it is inefficient.

The first two problems are unacceptable, illustrating why it is important to check that tables are in BCNF. It is important to know how to recognize and fix these problems, in order to:

- Avoid them in your own designs,
- Recognize and fix them in someone else's design
- Recognize and fix them in a designs that have been modified, and
- Recognize and fix them when porting data into a database from unnormalized sources such as spreadsheets.

Panel B in the figure illustrates how we fix these problems by decomposing the table into two tables, one representing students assigned to classes and the other representing instructors assigned to classes. Both tables satisfy BCNF.



Alternative Explanation

The Boyce-Codd condition means that any time that the value of any column can be determined from the values of any other set of columns then the other set of columns must be a candidate key. Let A be any column in a table, and let C be a different set of columns that determines A. Then if the table is in Boyce-Codd normal form, then C must determine not only A, but must also determine the values of all of the other columns in the table. This is a severe constraint, and it is just what we want, because it excludes all dependences except those based on candidate keys, without excluding the cases where there are multiple candidate keys. Today most commercial OLTP (Online Transaction Processing) databases are designed to Boyce-Codd normal form, though sometimes they are carefully denormalized afterwards, to improve performance.

Performance Tip

In this example the BCNF normalization may improve performance. There is one row in the table on the left in Panel B for each student, and one row in the table on the right for each section. Since there are normally many students per section, there are probably many more rows in the table on the left than there are in the table on the right. After the normalization the table on the left has one less column (STAFF_ID) so the rows are shorter, and the left hand table is significantly smaller after the normalization. As a result, the total size of the two tables in Panel B is probably less than the size of the one table in Panel A, so the normalization reduces database size. Smaller tables are more likely to be in the block buffer cache, so this reduces the amount of input and output to storage required to access the data in the tables. As a result, correcting the anomalies will probably also result in a faster database. There is another more advanced consideration. The right-hand table in Panel B is probably small enough so that it will be resident in the database block buffer cache, so that joining it on modern DBMS such as Oracle will not slow processing measurably.

Test Yourself

Is this table in Boyce-Codd normal form?

BookID – primary key

10DigitISBN

13DigitISBN

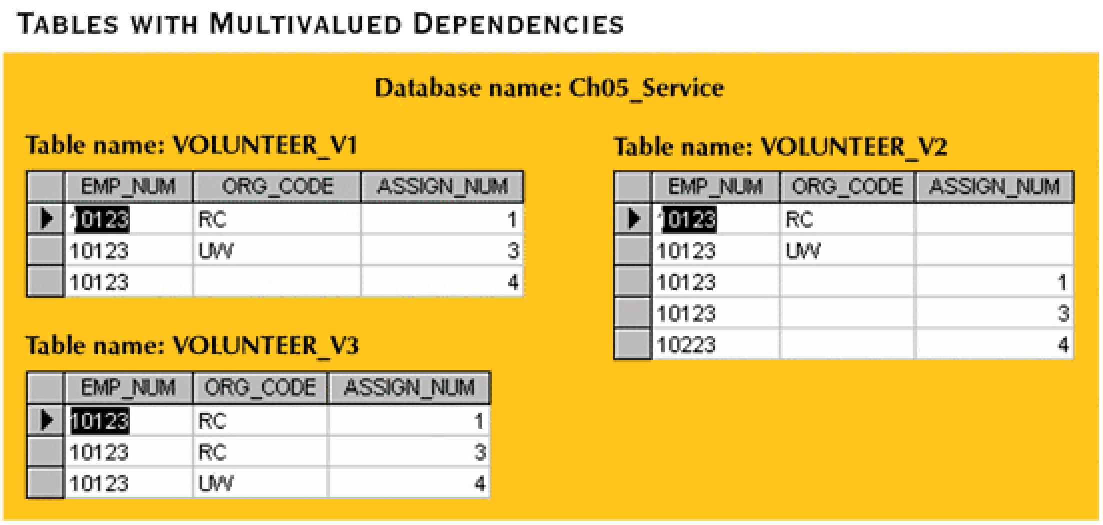
BookTitle

BookAuthor

Yes. All the determinants (BookID, 10DigitISBN, 13DigitISBN) can by themselves determine all the other fields in the row, so it satisfies the rule that all the determinants are candidate keys.

Fourth Normal Form

When a table represents a relationship between entities, it is important that each row in the table represents only one independently varying relationship between those entities. The fourth normal form checks this constraint. **A table is in fourth normal form (4NF) if it is in 3NF and has no multiple sets of multi-valued dependencies.** Suppose that we have a VOLUNTEER table that represents the facts of employees volunteering for organizations such as the Red Cross and United Way. This table represents the relationships between the employees and the organizations for which they volunteer. The following table illustrates three designs for the VOLUNTEER table. All three tables have the same columns; they differ in the presence of nulls.


TABLES WITH MULTIVALUED DEPENDENCIES

Database name: Ch05_Service

Table name: VOLUNTEER_V1

EMP_NUM	ORG_CODE	ASSIGN_NUM
10123	RC	1
10123	UW	3
10123		4

Table name: VOLUNTEER_V2

EMP_NUM	ORG_CODE	ASSIGN_NUM
10123	RC	
10123	UW	
10123		1
10123		3
10223		4

Table name: VOLUNTEER_V3

EMP_NUM	ORG_CODE	ASSIGN_NUM
10123	RC	1
10123	RC	3
10123	UW	4

The first design has a nullable ORG_CODE column, with NULLs present, and it has no primary key. The second table has nulls in both ORG_CODE and ASSIGN_NUM, so it doesn't have a primary key either. The third design has a primary key consisting of all three columns, but it is still not really correct, and it could take many rows to store all the combinations of relationships between employees, organizations, and assignments. All three tables are trying to represent both the relationship between the employee and organization and the relationship between the employee and their assignments in the company. Because these relationships are independent, trying to represent them in the same table creates problems. The following figure from the text illustrates how the relationship between employees and their volunteer organizations is represented in the SERVICE_V1 table and the relationship between employees and their assignments is represented in the ASSIGNMENT table.

A Set of Tables in 4NF

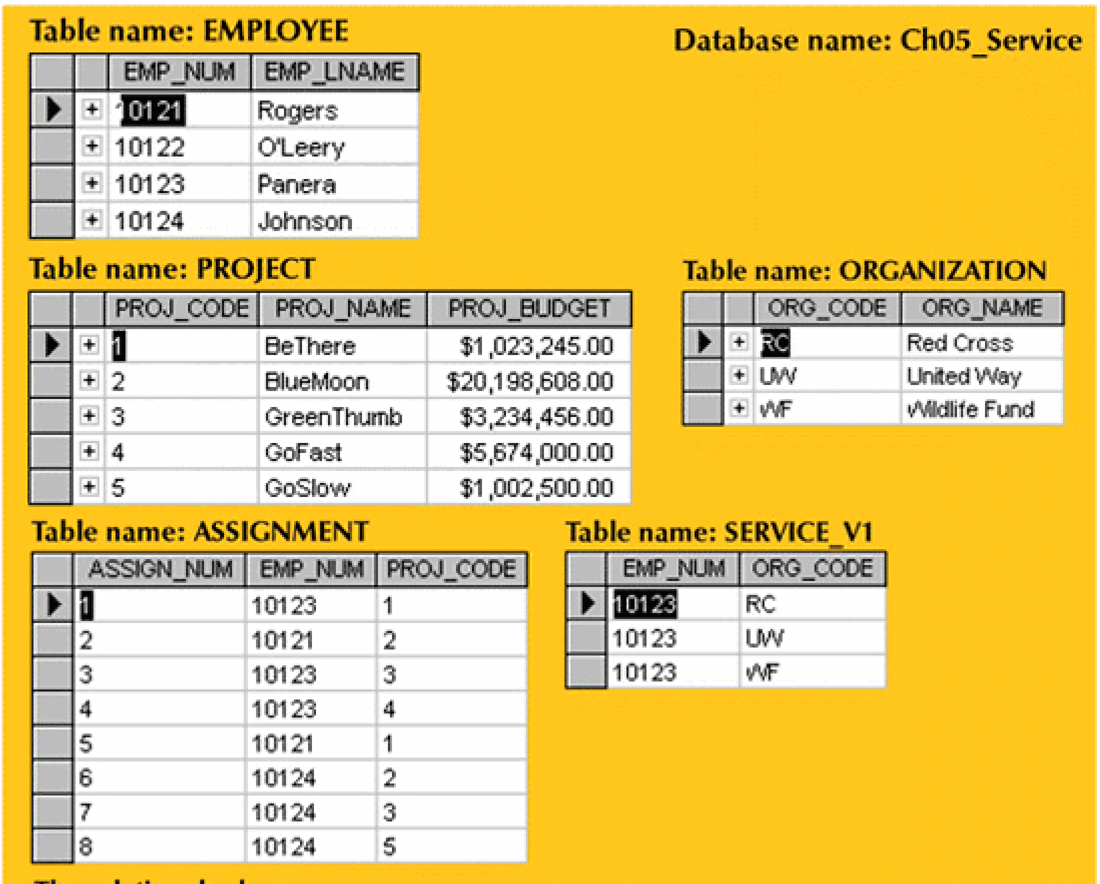


Table name: EMPLOYEE

Database name: Ch05_Service

EMP_NUM	EMP_LNAME
10121	Rogers
10122	O'Leery
10123	Panera
10124	Johnson

Table name: PROJECT

PROJ_CODE	PROJ_NAME	PROJ_BUDGET
1	BeThere	$1,023,245.00
2	BlueMoon	$20,198,608.00
3	GreenThumb	$3,234,456.00
4	GoFast	$5,674,000.00
5	GoSlow	$1,002,500.00

Table name: ORGANIZATION

ORG_CODE	ORG_NAME
RC	Red Cross
UW	United Way
WF	Wildlife Fund

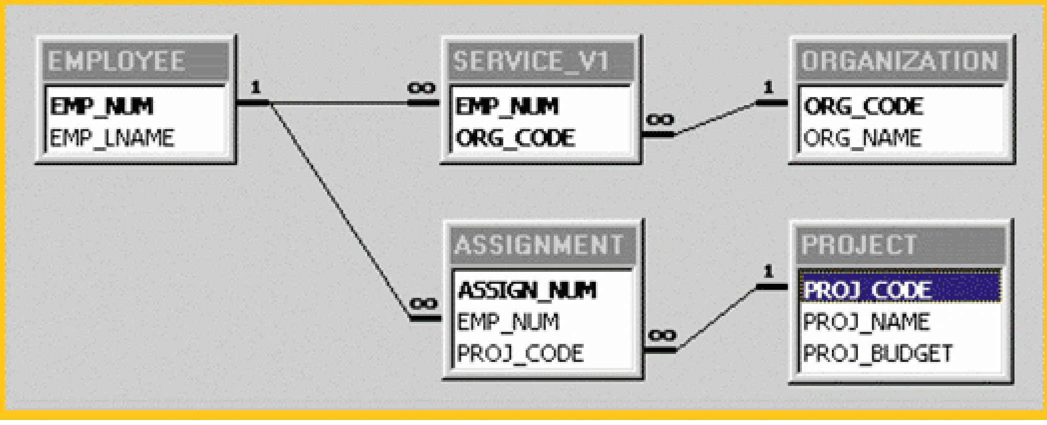
Table name: ASSIGNMENT

ASSIGN_NUM	EMP_NUM	PROJ_CODE
1	10123	1
2	10121	2
3	10123	3
4	10123	4
5	10121	1
6	10124	2
7	10124	3
8	10124	5

Table name: SERVICE_V1

EMP_NUM	ORG_CODE
10123	RC
10123	UW
10123	WF

The relational schema



The entity-relationship diagram at the bottom shows how the tables SERVICE_V1 and ASSIGNMENT associate the employees with their service organizations and projects respectively. All tables have primary keys. The primary key of SERVICE_V1 includes both of the columns. It is common for the primary key of an association table to include all of the columns of the table. The ASSIGNMENT table has two keys - the synthetic key ASSIGN_NUM and the natural key consisting of EMP_NUM and PROJ_CODE. This is also a common construction that is used when other tables need to refer to the relationship, and it is better to have a smaller foreign key than the natural key for the relationship.

Test Yourself

Consider the following table which represents employees and their assignments to different managers on different projects on different weeks. An employee can work on multiple projects with different managers in the same week. The table has a four-column primary key. A Manager may work on multiple projects, so there are no transitive dependencies and the table is in third normal form. Is this table in Fourth normal form?

EmployeeID – primary key
WeekEndingDate – primary key
ProjectCode – primary key
Manager – primary key
ProjectResponsibility
HoursWorked

This table is not in fourth normal form, because there is a multivalued dependency between ProjectCode and Manager. The underlying problem is that the design does not separate the management organization of the projects from the work done by employees on the projects. One fourth-normal-form design would explicitly represent ProjectArea (a group with one manager) and their managers. With this change the table would have a three-column primary key consisting of EmployeeID, WeekEndingData, and ProjectArea, with dependent columns ProjectResposibility and HoursWorked, and there would be no multivalued dependency.

Denormalization

As we have seen, the normalization process breaks down complex tables into multiple simpler tables that are usually easier to work with and that function without anomalies. Sometimes normalization increases the size of databases, and it usually increases the number of tables that must be joined in queries. When more tables must be joined there should usually be indexes to support those joins, as well, which increases database size. Because each table queried normally requires some I/O the larger number of tables also increases the number of I/O operations required to execute the queries. Because joins involving many tables sometimes have no efficient query plans, the processing time can increase dramatically when the number of joined tables increases. For these reasons the database design process involves compromises between normalization and database performance.

"Over-normalization" refers to what happens when the normalization process is carried on without regard to its consequences for the understandability of the schema, performance, database size, or the number of tables. Here are a few guidelines that you can use to question whether a normalization step may not be helpful:

How to tell when you may be over-normalizing

- If you have a table A that contains one representation of a conceptual object B, but object B is only ever present as part of object A, then there is not usually any advantage in separating B into its own table. You should be able to keep the columns that represent B as columns of table A, eliminating joins of B. Conversely, if there are many repetitions of the rows of object B in object A and the rows are long, then it may be faster to normalize.
- If you find yourself joining more than a few tables, and the queries are too slow, in spite of careful indexing, you should look for ways to combine tables. The number of tables that you can join depends on the size of the tables. Efficiently joining even two terabyte-class tables

takes great care. The best commercial databases can usually handle joins of up to about eight moderate-sized tables without too much degradation, when there is appropriate RAM and other resources, but fewer tables is usually much faster.

- When you find that the normalization has broken natural domain objects into unnatural pieces, it may be time to keep the less normalized form. Correct application of the normalization rules should not do this, so you may have made a mistake. Check the object rules presented on the next page. You may also consider reducing the number of tables by using one of the ANSI/ISO standard collection types in databases such as Oracle (Loney, Ch. 34) and DB2.
- When the normalization breaks up tables that correspond to object classes in object-oriented application software you should question whether this normalization is wise, and ask why the application object modelers did not so decompose their objects. Application operations with the database are usually fastest when the tables in the database correspond to application software object classes.

Denormalization is often carried out to improve the correspondence between the data access patterns and the way the data is stored. For example, I consulted on the design of a large data warehouse for a telephone company. The warehouse had a multi-billion-row CALL_DETAIL table with a row for each phone call. The table had a number of small columns that represented commonly accessed data such as the originating number, destination number, start time, and stop time. The table also had one very large text column that held detail about the routing of the call. This one column was many times larger than the sum of the size of all the other columns. The large column was rarely accessed. We moved this column to its own table, so that the data for this column would not be brought in from storage each time the frequently accessed data was accessed, slowing the performance of those frequent queries. The result was about an order of magnitude increase in average performance, with insignificant slowing of the queries that access the very large column, as expected.

The dimensional schemas used in data marts and data warehouses are technically denormalized according to the normal forms that we have discussed here, but I prefer to think of them as normalized in a very different way, according to a different set of normalization rules, which optimize for query performance and database size. We will cover these rules and dimensional design in Week 4.

Test Yourself

What are some of the reasons for denormalizing a schema? (Check all that are true.)

Denormalizing is important when the application could get better performance with fewer joins.

This is true.

Denormalizing can reduce the number of indexes required to efficiently support joins, and so reduce database size and index maintenance overhead.

This is true.

Denormalizing is important if the way the data is stored is not optimal for the way the data is accessed.

This is true.

Object-Oriented Design Tests

Definition

An *object class* refers to a kind of entity. Nouns which are not proper nouns are names for object classes. Proper nouns are names for instances of those classes. For example, *city* is a noun that is the English language name of an object class and *Boston* is the name of an instance of that class. CS682 *Systems Analysis and Design* covers this topic in detail.

What all of these normalization tests are really doing is checking that each table represents a different kind of object class (entity type), and not part of an object class, or parts of multiple object classes, or other constructs that we don't normally want in our databases. What I would like to share with you now is a different set of rules and way of thinking that will help you identify the objects in your database. These rules and ways of thinking don't replace the normalization rules that we have studied, but supplement them. I have found that when these object checks succeed and the normalization checks succeed, that the design is quite solid. The object-oriented ways of thinking also help us when we are designing databases to support applications developed using object-oriented languages such as Java, C++ or C#.

The object-oriented tests depend on thinking about the real-world entities represented by the rows in the table. Many real-world entities are physical objects such as airplanes or shoes. Other entities, such as accounts and degrees, are not primarily physical, but they gain a physical-like importance and permanence because people ascribe to them the reality of physical objects.

- The first test is whether all parts (columns) of the entities represented by the rows in this table are created and destroyed at the same time. If some parts can be created at different times, this suggests that those parts may really represent a different object, which is often best represented in a different table, often linked by a foreign key. Pay particular attention to nullable columns to make sure that they are really part of the same object as the other columns.
- A second test is whether the entity type represented by this table has a natural name. If the entities represented by the rows all share a common type name, then the table probably represents a natural object. The name may be a phrase. Many natural object types have names, but be aware that not all natural object types have names, even esoteric names known only to experts in the problem domain. If you encounter one of these unnamed object types you can test it by asking domain experts if they recognize it. They may say something like, "Sure, it's something that we all understand; we just don't have a name for it."
- Do all of the entities represented by the rows in this table exhibit the same behavior? If the behaviors of different entities are different this suggests that they may be of different types.

As a general rule in production databases it is best to use tables that are normalized to at least BCNF, and use denormalization sparingly. If you denormalize you should understand why you are doing it, and make sure that you have analyzed and dealt with the anomalies and potential data inconsistencies that may result. A different set of general rules applies to analysis databases such as data warehouses.

Test Yourself

You're considering a database design for an online candy store. Apply the rules listed above and name some of the possible objects in your design. (Check all that are true.)

Candy

This is true.

Price

This is false. Price is more likely to be an attribute.

Customer

This is true.

Order

This is true.

Lecture 1B - Database Programming

Overview

All industrial-strength DBMS include a database programming language. PL/SQL is the programming language developed by Oracle. Database programming languages such as TransactSQL, which is supported by Microsoft SQL Server and Sybase, are similar. Database programs are used to write triggers, stored procedures, stored functions, and object methods. Database programming is important for databases that support large numbers of applications, for advanced database security, for efficient execution of conditional operations, and for data and compute intensive operations such as loading data warehouses.

Learning Objectives

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Describe what database programming is and how it is used.
- Write and run programmatic SQL using Oracle's PL/SQL or Microsoft SQL Server's Transact SQLanonymous PL/SQL blocks using SQLPlus.
- Write and test the common kinds of database triggers in Oracle's PL/SQL or Transact SQL.
- Write and test stored procedures in PL/SQL or Transact SQL.

- Write and test stored functions in PL/SQL or Transact SQL.
- Write and test code to catch and handle database exceptions.

You will also be prepared to use PL/SQL to write object methods when we discuss object-relational databases.

Introduction to Database Programming with PL/SQL

Database programming languages combine SQL, parameter passing, execution control constructs and exception handling to create powerful database-oriented programming languages. Database programs are sent to the database as text only once, where they are parsed, compiled, optimized, and stored as an integral part of the database. The database programs may then be executed many times without incurring this overhead.

There are three common kinds of database programs—stored procedures, stored functions, and triggers. Object-relational DBMS such as Oracle and DB2 also support methods, which database programs associated with object types. We will learn to write and run all three common types of database programs using PL/SQL or Transact SQL. In a later lecture on object-oriented extensions we will also learn how to write methods in PL/SQL.

Database programming meets important business needs, including:

- Encapsulating common functionality in a way that can be easily shared between applications, thereby minimizing application size and complexity, minimizing life cycle costs and improving quality.
- Improving performance by eliminating repeated SQL transmissions, parsing, and optimization.
- Reducing the need to communicate intermediate results with applications, thereby improving speed and reducing network traffic.
- Providing the fastest way to implement data-intensive database operations. Database code has minimal data access overhead.
- Supporting triggers, which is database code that runs automatically when specified operations change the database. Triggers can be used for many things, including enforcing integrity constraints and maintaining denormalized data.
- Providing the opportunity to not support direct access to the underlying tables, and allowing access only through stored procedures. The stored procedures can then enforce arbitrary security, audit, or other functionality. The stored procedures serve as an interface layer, hiding underlying database changes from applications.
- If database access is only through stored procedures, then the stored procedures can enforce integrity constraints. Stored procedures can support complex transaction-oriented business constraints, and they are usually faster than implementing constraints using more local constraint mechanisms such as foreign keys or triggers. Stored procedures are used for this reason in very high performance systems and systems with complex integrity constraints.

So, database programming is very useful, particularly when there are many applications, complex integrity constraints, high throughput, and/or critical databases. For these reasons database programming is widely used for enterprise and other critical databases.

Test Yourself

What are some of the reasons to use stored procedures? (Check all that are true.)

Stored procedures can be reused between applications.

This is true.

Stored procedures can improve performance.

This is true.

Stored procedures provide additional functionality.

This is true.

Stored procedures execute data intensive applications more slowly than procedural code executing on database clients.

This is almost always false. Stored procedures execute in the DBMS with direct access to the data, eliminating the need to encode, transmit, and decode the data. Stored procedures execute on a database server, which is almost always more capable than database client platforms.

Language and Compatibility

To say that the languages are *semantically* similar means that users can say things in the different languages that mean about the same thing. This is not surprising, because the different vendors designed their database programming languages to solve the same customer problems.

To say that the languages are *syntactically* incompatible is to say that the grammars for the languages are sufficiently different that programs written in one language are not syntactically (grammatically) correct in the other languages.

In the 1980s DBMS vendors, including Oracle, IBM and Sybase implemented database programming languages because their customers needed them, and the ANSI standards committee did not address database programming until ANSI SQL 1999. As a result there are different *semantically* similar but *syntactically* incompatible database programming languages. There are programs that translate database code from one DBMS to another. The translated code may not function in precisely the same way, often because of differences in the underlying databases, so database programmers need to be attentive when porting database programs. ANSI SQL 99 includes a standard database programming language based on the language in IBM's DB2, but it is not yet widely supported. DB2 also supports PL/SQL.

Footnote: MySQL's database programming language is newly implemented, and it is based on the ANSI SQL99 standard.

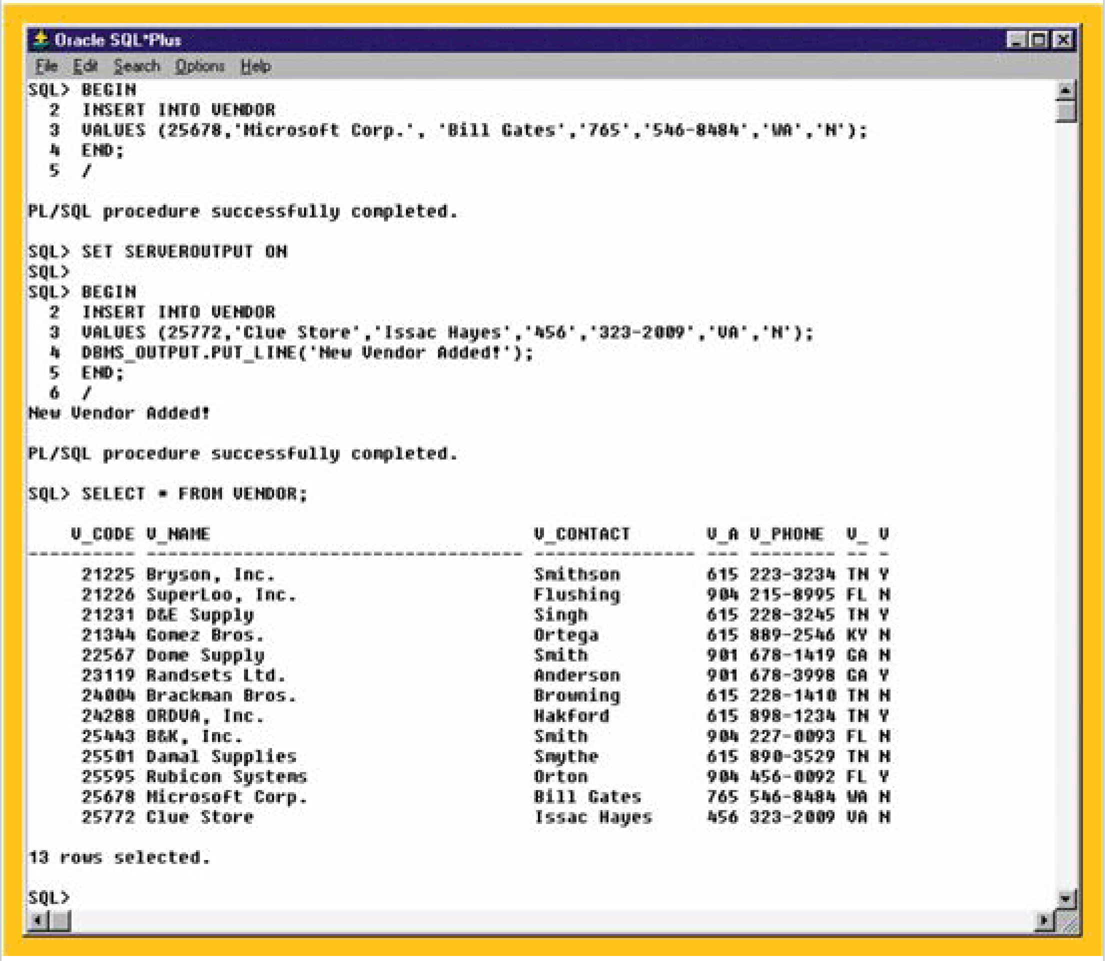
What is database programming? Database programming involves the definition, coding, testing, and maintenance of triggers, stored procedures, stored functions, and methods.

Database programming is also a specialized software engineering job category that involves significant database and programming skills.

PL/SQL Blocks

The most basic PL/SQL construct is a PL/SQL block. A PL/SQL block is a piece of code that is sent to the database, where it is executed just once, and not saved for future use the way a stored procedure or function is stored. A PL/SQL block is *anonymous*, meaning that it has no name. A PL/SQL block can't be called again, so we don't need to give it a name by which we can be refer to it in future calls. A PL/SQL block can begin with the word DECLARE or with the word BEGIN, and it ends with the word END. The following figure from the text illustrates some very simple PL/SQL blocks.

Anonymous PL/SQL Block Examples



```
SQL> BEGIN
     INSERT INTO VENDOR
     VALUES (25678,'Microsoft Corp.', 'Bill Gates','765','546-8484','WA','N');
     END;
     /

PL/SQL procedure successfully completed.

SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
     INSERT INTO VENDOR
     VALUES (25772,'Clue Store','Issac Hayes','456','323-2009','VA','N');
     DBMS_OUTPUT.PUT_LINE('New Vendor Added!');
     END;
     /
New Vendor Added!

PL/SQL procedure successfully completed.

SQL> SELECT * FROM VENDOR;

    V_CODE V_NAME                              V_CONTACT       V_A V_PHONE  V_ V
---------- ----------------------------------- --------------- --- -------- -- -
     21225 Bryson, Inc.                        Smithson        615 223-3234 TN Y
     21226 SuperLoo, Inc.                      Flushing        904 215-8995 FL N
     21231 D&E Supply                          Singh           615 228-3245 TN Y
     21344 Gomez Bros.                         Ortega          615 889-2546 KY N
     22567 Dome Supply                         Smith           901 678-1419 GA N
     23119 Randsets Ltd.                       Anderson        901 678-3998 GA Y
     24004 Brackman Bros.                      Browning        615 228-1410 TN N
     24288 ORDVA, Inc.                         Hakford         615 898-1234 TN Y
     25443 B&K, Inc.                           Smith           904 227-0093 FL N
     25501 Damal Supplies                      Smythe          615 890-3529 TN N
     25595 Rubicon Systems                     Orton           904 456-0092 FL Y
     25678 Microsoft Corp.                     Bill Gates      765 546-8484 WA N
     25772 Clue Store                          Issac Hayes     456 323-2009 VA N

13 rows selected.

SQL>
```

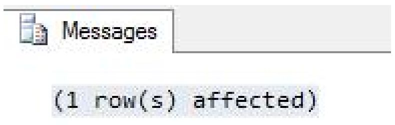
The example at the top just illustrates that Oracle will accept a single INSERT statement even if it is preceded by BEGIN and followed by END, and that when the block has executed Oracle and SQLPlus tell us that a procedure has successfully completed, even though it is not a procedure but just a PL/SQL block. The second example shows us that a PL/SQL block can include a call to the PUT_LINE procedure in the DBMS_OUTPUT built-in package. Such a procedure call is not legal as a SQL statement, so this tells us that we can do more inside of a PL/SQL block than we can in just SQL. Note the line SET SERVEROUTPUT ON. This enables the output of the DBMS_OUTPUT package. Without this line the block would have executed just fine, but the line "New Vendor Added!" would not have appeared in SQLPlus or another database client.

Transact-SQL Block Example

A Transact SQL (T-SQL) block can begin with the word DECLARE or with the word BEGIN, and it ends with the word END. The following illustrates a very simple T-SQL block.

```
begin
insert into VENDOR
VALUES (25678, 'Microsoft Corp.', 'Bill Gates', '765', '546-8484',
'WA','N');
END;
```

When it completes successfully, you'll see the following under "Messages":

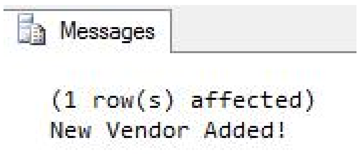


```
(1 row(s) affected)
```

The equivalent for Oracle's DBMS_OUTPUT.PUT_LINE in T-SQL is the command, *PRINT*. So to print a message in T-SQL after the INSERT command, you could use this block of code:

```
begin
insert into VENDOR VALUES (25772, 'Clue Store', 'Isaac Hayes', '456', '323-
2009', 'VA','N');
PRINT 'New Vendor Added!';
END;
```

When this is executed in T-SQL, the resulting message would look something like this:



```
(1 row(s) affected)
New Vendor Added!
```

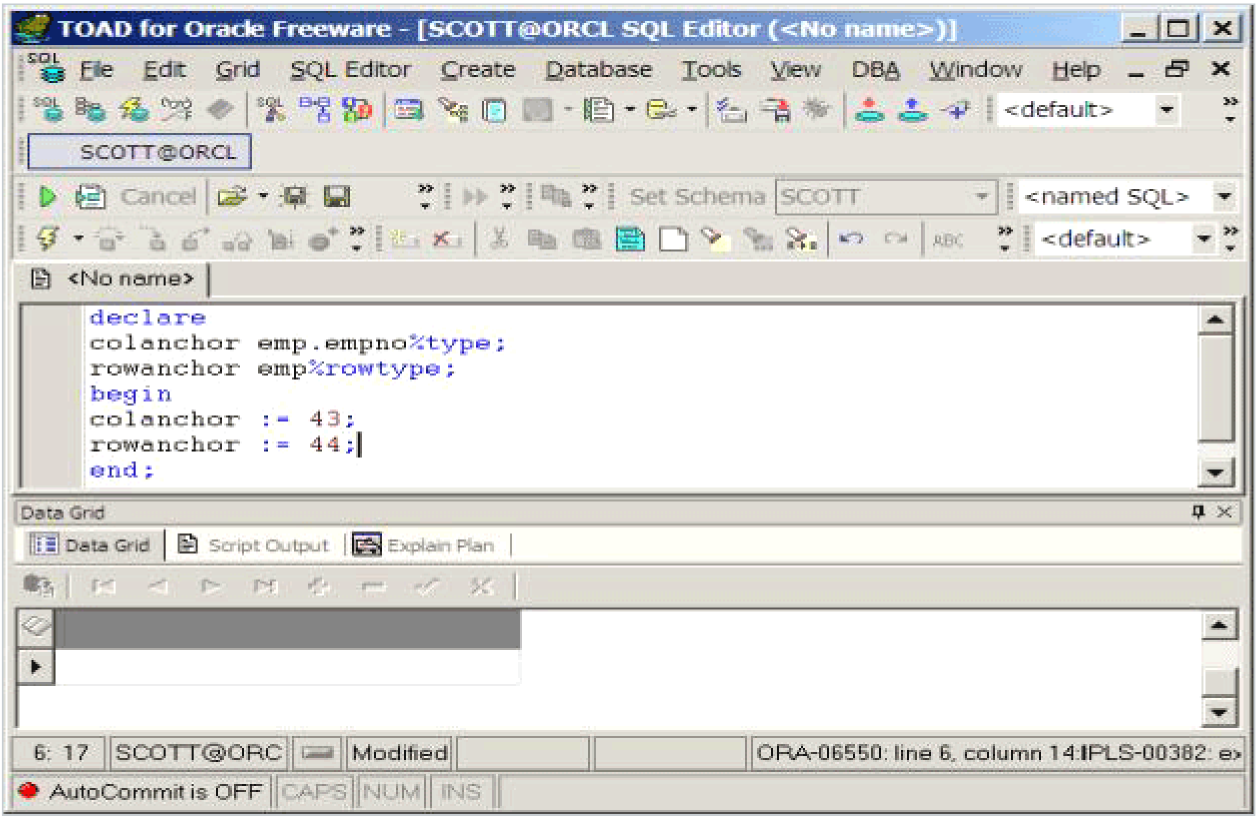
Note

In the first Oracle example, creating an anonymous block will have the same result as entering the INSERT statement on the command line by itself, because there are no statements inside the block other than the INSERT statement.

In the second example, however, using the block allows execution of the DBMS_OUTPUT.PUT_LINE command. If you try to execute the DBMS_OUTPUT.PUT_LINE directly from the SQL*Plus command line, you'll see the error, "Unrecognized command", but within the block it works as shown in the example.

PL/SQL Block with a DECLARE Section

We now examine the full form of a PL/SQL block with a DECLARE section:


TOAD for Oracle Freeware - [SCOTT@ORCL SQL Editor (<No name>)]
File Edit Grid SQL Editor Create Database Tools View DBA Window Help
SCOTT@ORCL
Cancel
Set Schema
SCOTT
<named SQL>
<default>
<No name>
Data Grid
Script Output
Explain Plan

```
declare
colanchor emp.empno%type;
rowanchor emp%rowtype;
begin
colanchor := 43;
rowanchor := 44;
end;
```

```
6: 17   SCOTT@ORC   Modified   ORA-06550: line 6, column 14:IPLS-00382: e>
AutoCommit is OFF   CAPS  NUM  INS
```

Definitions of Declaration and Block

A ***declaration*** is a code segment that defines a symbol, such as *colanchor* above, and gives it a meaning. Usually declarations create variables.

In programming languages the term ***block*** describes a segment of code that contains declarations and statements and perhaps exception handling. Depending on the programming language, blocks are delimited by a symbol such as BEGIN or left brace, and ended by a symbol such as END or right brace.

PL/SQL Maintainability Tip

When a data type in the database changes it is often necessary to change the data types in the PL/SQL or other database code. Oracle provides the %TYPE and %ROWTYPE syntax to permit you to anchor your PL/SQL data types to the corresponding data types in the database. Sometimes it is helpful to anchor our declarations to something closer to a schema type than a column type. (This is similar to using an abstract data type, which we will study in the lecture on object-relational database features.) This can be done by creating a special table with one column representing each business-oriented data type. This table typically has no data, and is used solely to anchor declarations. For example, this table might have a first_name column of type VARCHAR(20), an account_balance column of type NUMBER(9,2) and a gender column of type CHAR(1). Database CASE tools such as

Oracle Designer support schema-level data types, and these tools can be used to create this data types anchoring table, as well as to implement coordinated schema data type changes.

As one might expect, the DECLARE section before the BEGIN is where you can put declarations for the block. This example illustrates three additional PL/SQL features. One of these features is the anchoring of the data types of variables to the corresponding data types in the database. Consider the line of code:

```
colanchor emp.empno%type;
```

This line of code creates a new symbol *colanchor* which has as its scope the PL/SQL block. The type of this new symbol is declared to be the same as the type of the empno column of the emp table. The type of the variable colanchor is *anchored* to the type of the empno column. We don't need to know the type of the empno column to create the new variable to hold values of the empno column; we just anchor the declaration to the column. This anchoring of variables is very useful in improving the maintainability of PL/SQL code. Large enterprise database systems may have tens to hundreds of thousands of lines of PL/SQL code. If this code is built using anchoring, and the data types in the underlying schema change, then all that the developer has to do is recompile and appropriately retest the PL/SQL code, rather than laboriously searching the PL/SQL source code for all of the places where the declarations need to change to keep up with the schema changes.

Notice the second declaration seen above:

```
rowanchor emp%rowtype
```

This declaration creates a new variable *rowanchor* which has as its data type a record with the same structure as an entire row of the emp table. As we will see later, such records are useful as a place to store data in row-by-row processing with explicit cursors.

Note the line of code near the bottom of the PL/SQL block above:

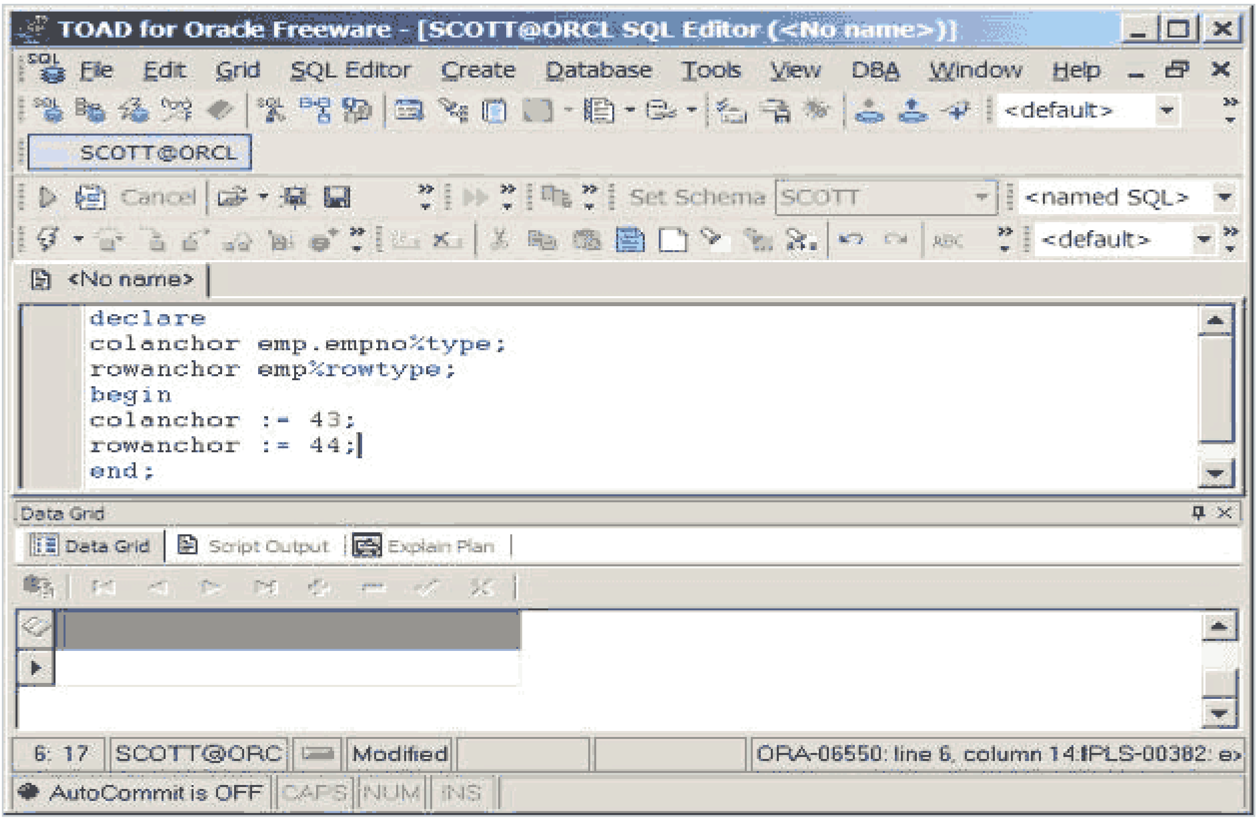
```
rowanchor := 44;
```

This is just a simple assignment statement. The ":=" is a two-character symbol that represents the assignment of a value to a variable in PL/SQL. (This has the same meaning as the "=" symbol in Java or C++.) What this statement does is give the local variable *rowanchor* the value 44. You will probably notice that this is an error. It makes no sense to assign a simple value to a whole record which contains multiple simple values. The correct syntax would be something like rowanchor.empno := 44;

Note from the block in the lower right hand corner of the TOAD window (shown above) that the PL/SQL block generated an error message.

TOAD Errors

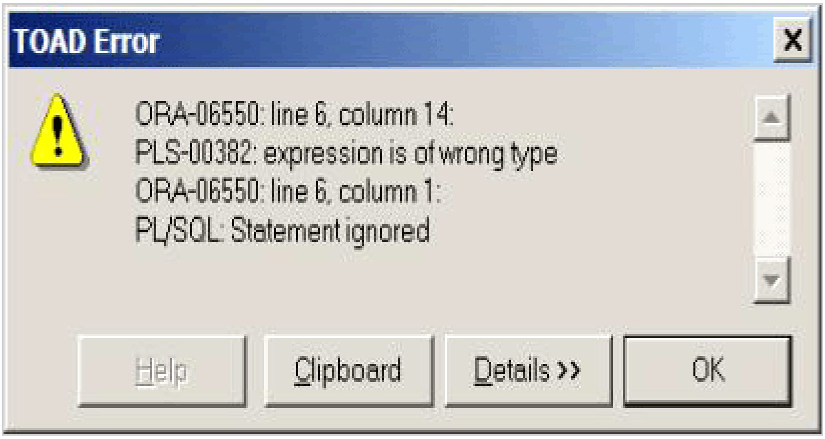
Now let's generate an error to see how TOAD presents errors to us:


TOAD for Oracle Freeware - [SCOTT@ORCL SQL Editor (<No name>)]
File Edit Grid SQL Editor Create Database Tools View DBA Window Help
SCOTT@ORCL
Set Schema SCOTT
<named SQL>
<No name>
Data Grid
Script Output
Explain Plan

```
declare
colanchor emp.empno%type;
rowanchor emp%rowtype;
begin
colanchor := 43;
rowanchor := 44;
end;
```

```
6: 17 | SCOTT@ORC | Modified | ORA-06550: line 6, column 14:PLS-00382: e>
AutoCommit is OFF | CAPS | NUM | INS
```

The error that I made is to attempt to assign the simple value 44 to the rowanchor variable, which is a record anchored to the table emp. Oracle returns an error and TOAD presents us with a popup:



```
ORA-06550: line 6, column 14:
PLS-00382: expression is of wrong type
ORA-06550: line 6, column 1:
PL/SQL: Statement ignored
```

Notice that there is more than one error message. This is typical of Oracle and other RDBMS. Usually the error message that we care about is near the top; in this case it is the second, PLS error that tells us that the expression is of the wrong type.

When something more complicated is going on, Oracle may give you a dozen or more error messages from one real error. For example, suppose that a utility stored procedure has an unhandled SQL error and that stored procedure is invoked from a trigger that was “triggered” by another SQL statement in a transactional stored procedure. Then you would expect error messages from the SQL statement, from the utility stored procedure containing the SQL statement, from the trigger, from the SQL statement that caused the trigger, and from the transactional stored procedure.

Note, to assign a value to rowanchor, you will need to use a cursor to read through the rows and populate the rowanchor.

```
CURSOR c1 IS
SELECT empno, ename FROM emp;
```

Then it will be possible to assign a value to one of the columns represented by rowanchor. The following illustrates the corrected code:

```
declare
colanchor emp.empno%type;
rowanchor emp%rowtype;
CURSOR c1 IS
SELECT empno, ename FROM emp;
begin
colanchor:=43;
rowanchor.empno:=44;
end;
```

Test Yourself

When would you use an anchor for a declared variable in a PL/SQL block?

Use a row anchor (%rowtype) for one attribute when you need to perform a join.

This is false. Row anchors are used to declare records, not single attributes.

Use a column anchor for a variable whose type may change over the lifetime of the database.

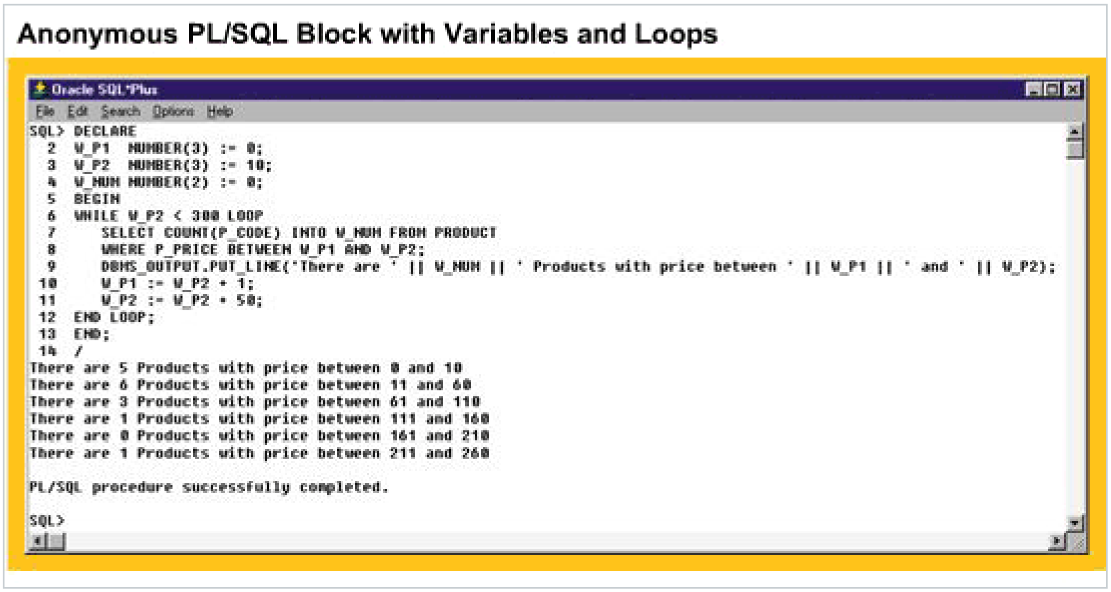
This is true. This design greatly reduces the effort to update the data types in persistent stored modules such as stored procedures.

Use a row anchor to declare a record which has the same structure as a table in the database.

This is true. Anchoring record declarations to database tables can greatly simplify database code maintenance and record-level operations.

A More Complex Example

We now examine a PL/SQL block of more interesting complexity, from the text:

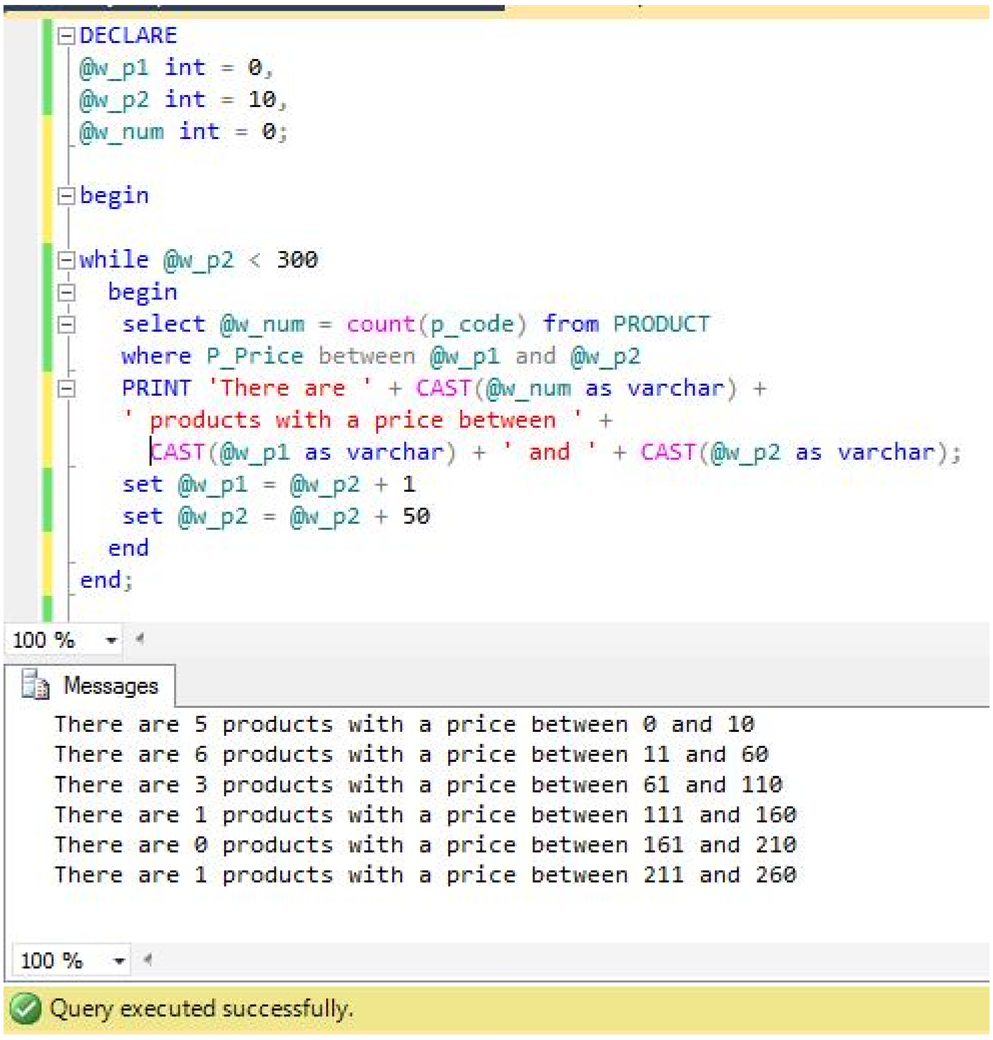


Anonymous PL/SQL Block with Variables and Loops

```
SQL> DECLARE
  2  W_P1  NUMBER(3) := 0;
  3  W_P2  NUMBER(3) := 10;
  4  W_NUM NUMBER(2) := 0;
  5  BEGIN
  6  WHILE W_P2 < 300 LOOP
  7     SELECT COUNT(P_CODE) INTO W_NUM FROM PRODUCT
  8     WHERE P_PRICE BETWEEN W_P1 AND W_P2;
  9     DBMS_OUTPUT.PUT_LINE('There are ' || W_NUM || ' Products with price between ' || W_P1 || ' and ' || W_P2);
 10     W_P1 := W_P2 + 1;
 11     W_P2 := W_P2 + 50;
 12  END LOOP;
 13  END;
 14  /
There are 5 Products with price between 0 and 10
There are 6 Products with price between 11 and 60
There are 3 Products with price between 61 and 110
There are 1 Products with price between 111 and 160
There are 0 Products with price between 161 and 210
There are 1 Products with price between 211 and 260

PL/SQL procedure successfully completed.

SQL>
```

Note that the variables declared in the declarations section are initialized; this is a good practice. Notice the SELECT INTO syntax. There is no standard output such as the user screen for PL/SQL to select a result set into, so SELECT statements in PL/SQL must always have the INTO clause. Notice the BETWEEN clause; this is shorthand for two WHERE clauses specifying the upper and lower bounds. Notice the two vertical bar characters in line 9; these two characters together comprise the SQL string concatenation operation. If you have been attentive you will realize that this session must have executed SET SERVEROUTPUT ON before this code was executed, because the DBMS_OUTPUT.PUT_LINE procedure results in output to the SQLPlus screen. Notice the PL/SQL WHILE loop, which executes everything between the LOOP and END LOOP as long as the condition (*W_P2* < *300* in this case), is true.

T-SQL Example



```
DECLARE
@w_p1 int = 0,
@w_p2 int = 10,
@w_num int = 0;

begin

while @w_p2 < 300
  begin
   select @w_num = count(p_code) from PRODUCT
   where P_Price between @w_p1 and @w_p2
   PRINT 'There are ' + CAST(@w_num as varchar) +
   ' products with a price between ' +
     CAST(@w_p1 as varchar) + ' and ' + CAST(@w_p2 as varchar);
   set @w_p1 = @w_p2 + 1
   set @w_p2 = @w_p2 + 50
  end
end;
```

```
There are 5 products with a price between 0 and 10
There are 6 products with a price between 11 and 60
There are 3 products with a price between 61 and 110
There are 1 products with a price between 111 and 160
There are 0 products with a price between 161 and 210
There are 1 products with a price between 211 and 260
```

Query executed successfully.

You might notice a few changes between the PL/SQL version and this T-SQL version.
In T-SQL:

1. All the variables used within the block are prefaced with an @ sign.
2. The WHILE construct is essentially the same. Use BEGIN for T-SQL to start the actions performed as long as the WHILE condition is true, and end the WHILE loop with an END statement before it rechecks the condition in the WHILE clause.
3. The SELECT…INTO construct is not valid in SQL Server, so instead, the SELECT statement below sets the variable @w_num equal to the value being selected, in this case count(p_code).
4. The PRINT statement in SQL Server performs the same function as DBMS_OUTPUT.PUT_LINE in PL/SQL. Instead of the double bars between clauses, SQL Server uses the plus (+) sign. Also, variables need to use the CAST command to allow the variable to be output as a string.
5. To establish a new value for a variable use the command SET, like this: set @w_p1 = @w_p2 + 1.

A List of Data Types

So far we have declared variables of NUMBER types and of one anchored record type. Let's examine all of the different types that can be declared in PL/SQL. The following tabular list of data types is derived from *Oracle:*

The Complete Reference.

BINARY_DOUBLE	FLOAT	NUMBER	SMALLINT
BINARY_FLOAT	INT	NUMERIC	STRING
BINARY_INTEGER	INTEGER	NVARCHAR2	TIMESTAMP
BOOLEAN	INTERVAL DAY TO SECOND	PLS_INTEGER	TIMESTAMP WITH LOCAL TIMEZONE
CHAR	INTERVAL YEAR TO MONTH	POSITIVE	TIMESTAMP WITH TIMEZONE
CHARACTER	LONG	POSITIVEN	UROWID
DATE	LONG RAW	RAW	VARCHAR
DEC	NATURAL	REAL	VARCHAR2
DECIMAL	NATURALN	ROWID	
DOUBLE PRECISION	NCHAR	SIGNTYPE	

This list includes the simple scalar PL/SQL data types. There are additional PL/SQL data types such as CURSOR and RECORD which declare things other than simple scalar data types. Additional non-scalar data types are summarized in the following table.

Composite Types	RECORD, TABLE, VARRAY
Reference Types	REF CURSOR, REF <object type>
LOB Types	BLOB, BFILE, CLOB, NCLOB

T-SQL Data Types

The following tabular list of data types for SQL Server 2012 is derived from the *technet.microsoft.com* website.

Exact Numerics

bigint	numeric

bit	smallint
decimal	smallmoney
int	tinyint
money	

Approximate Numerics

float	real

Date and Time

date	datetimeoffset
datetime2	smalldatetime
datetime	time

Character Strings

char	varchar
text	

Unicode Character Strings

nchar	nvarchar
ntext	

Binary Strings

binary	varbinary
image	

Other Data Types

cursor	timestamp
hierarchyid	uniqueidentifier
sql_variant	xml
table	

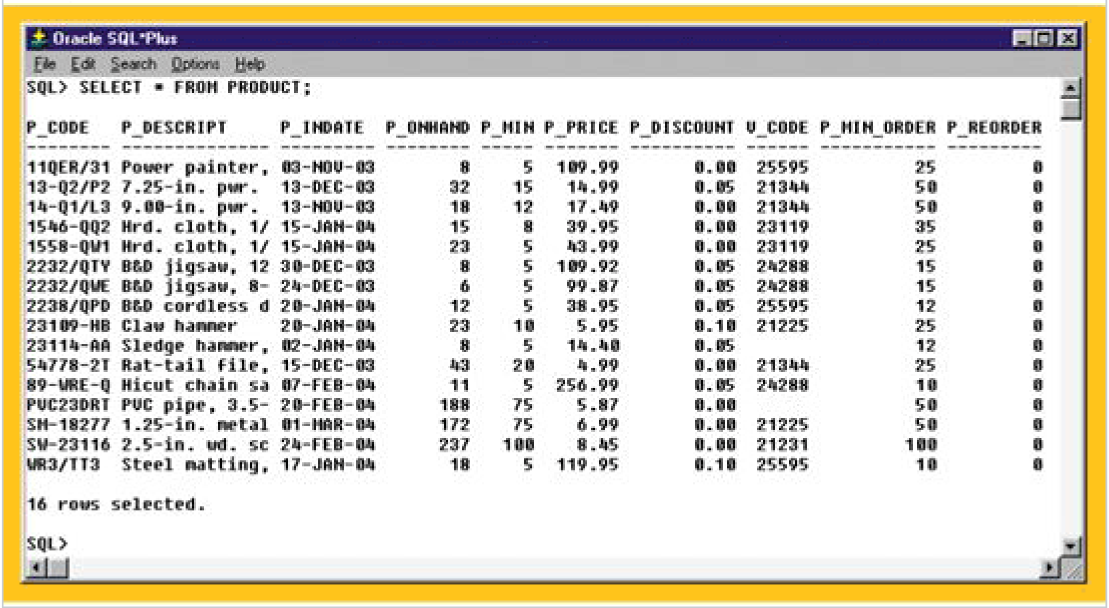
Triggers

A *trigger* is a kind of durable database object that you can develop using database programming languages such as PL/SQL and T-SQL. Triggers are automatically invoked by the DBMS when a triggering event associated with the trigger occurs. Triggering events may be DML statements such as inserts into a particular table or an update of a particular column in a table. In Oracle there are also DDL triggers, which are associated with the creation or deletion of database objects. Oracle also has login triggers, which are associated with the creation of a session, and database triggers associated with the startup of the database itself. We will cover each of these types of triggers in this section.

DML Triggers

A DML trigger is database code that executes when specified DELETE, INSERT, or UPDATE operations are performed. I will now show you how a trigger can be used to maintain denormalized data. I will use the examples from the text, so as to not to overload you with examples, but I will provide independent commentary. Let's begin by looking at the PRODUCT table, which includes denormalization:

The PRODUCT Table



```
SQL> SELECT * FROM PRODUCT;

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter,  03-NOV-03        8     5  109.99       0.00  25595          25         0
13-Q2/P2 7.25-in. pwr.   13-DEC-03       32    15   14.99       0.05  21344          50         0
14-Q1/L3 9.00-in. pwr.   13-NOV-03       18    12   17.49       0.00  21344          50         0
1546-QQ2 Hrd. cloth, 1/  15-JAN-04       15     8   39.95       0.00  23119          35         0
1558-QW1 Hrd. cloth, 1/  15-JAN-04       23     5   43.99       0.00  23119          25         0
2232/QTY B&D jigsaw, 12  30-DEC-03        8     5  109.92       0.05  24288          15         0
2232/QWE B&D jigsaw, 8-  24-DEC-03        6     5   99.87       0.05  24288          15         0
2238/QPD B&D cordless d  20-JAN-04       12     5   38.95       0.05  25595          12         0
23109-HB Claw hammer     20-JAN-04       23    10    5.95       0.10  21225          25         0
23114-AA Sledge hammer,  02-JAN-04        8     5   14.40       0.05                 12         0
54778-2T Rat-tail file,  15-DEC-03       43    20    4.99       0.00  21344          25         0
89-WRE-Q Hicut chain sa  07-FEB-04       11     5  256.99       0.05  24288          10         0
PVC23DRT PVC pipe, 3.5-  20-FEB-04      188    75    5.87       0.00                 50         0
SM-18277 1.25-in. metal  01-MAR-04      172    75    6.99       0.00  21225          50         0
SW-23116 2.5-in. wd. sc  24-FEB-04      237   100    8.45       0.00  21231         100         0
WR3/TT3  Steel matting,  17-JAN-04       18     5  119.95       0.10  25595          10         0

16 rows selected.

SQL>
```

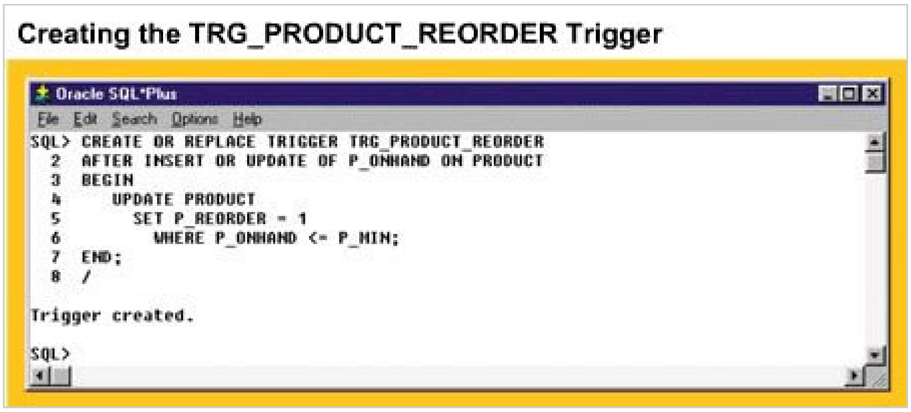
Boolean Expression

A Boolean expression is an expression which has as its value either *true* or *false*.

Notice the P_REORDER column. This column is intended to be one when P_ONHAND is less than P_MIN, and zero otherwise. Since P_ONHAND is greater than P_MIN for all rows in this table, P_REORDER is zero as it should be. This is included to illustrate the use of a trigger to maintain denormalizations. One would not normally include such a column in a product table, because it would increase the row length, and would so likely slow even the queries intended to identify products to reorder. P_REORDER is an integer-encoded Boolean equivalent to the *Boolean expression* (P_ONHAND < P_MIN). Most of the cost of evaluating such an expression is the cost of fetching the row from storage, and you would be hard pressed to measure the CPU cost. If this table had many millions of rows you might consider having such a P_REORDER column to have something to index, but in Oracle you would do better to implement this as a function-based index rather than a column plus an index. We will cover function-based indexes in the second lecture of week 2.

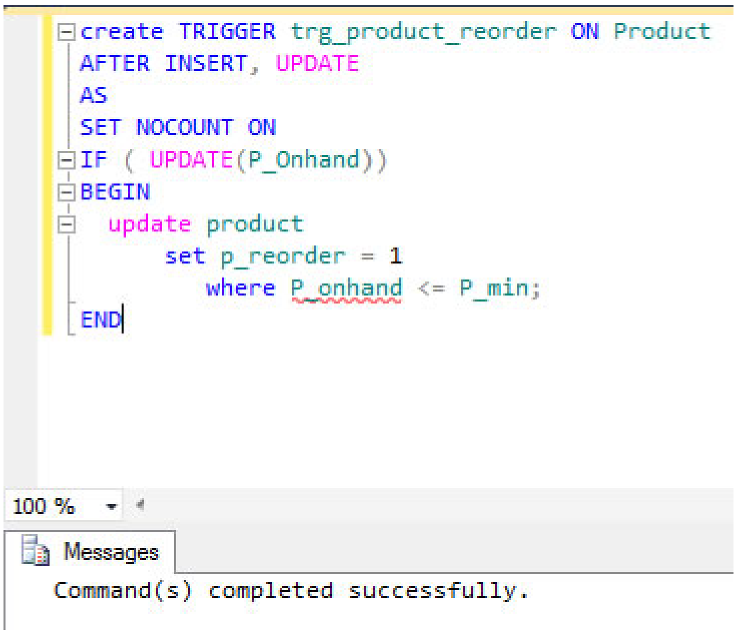
Triggers to Maintain Denormalization

Now let's look at the authors' demo first try at a trigger to maintain the P_REORDER denormalization


Creating the TRG_PRODUCT_REORDER Trigger

```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
  2  AFTER INSERT OR UPDATE OF P_ONHAND ON PRODUCT
  3  BEGIN
  4     UPDATE PRODUCT
  5        SET P_REORDER = 1
  6          WHERE P_ONHAND <= P_MIN;
  7  END;
  8  /

Trigger created.

SQL>
```

T-SQL Example



```
create TRIGGER trg_product_reorder ON Product
AFTER INSERT, UPDATE
AS
SET NOCOUNT ON
IF ( UPDATE(P_Onhand))
BEGIN
  update product
        set p_reorder = 1
           where P_onhand <= P_min;
END
```

100 %

Messages

```
Command(s) completed successfully.
```

Notice the use of CREATE OR REPLACE instead of just CREATE TRIGGER. This is a good practice in scripts that create durable database objects for Oracle databases, because what we want to do is create this particular trigger and we want the new one to replace an old one if it exists. If we had used just CREATE TRIGGER it would be an error if the trigger exists. This same principle applies to creating other durable objects such as stored procedures. There is no corresponding CREATE OR REPLACE TABLE syntax, because there may be data in the table, so DBMS vendors force us to explicitly say DROP TABLE or ALTER TABLE.

Notice the name of the trigger TRG_PRODUCT_REORDER. A better convention is to begin the trigger name with the name of the table that the DML trigger is associated with. By the industry convention this trigger would be named something like PRODUCT_REORDER_TR. The reason to begin the trigger name with its table name is so that when we list triggers ordering by name then the triggers for a table are listed together. The other reason is that trigger error messages can be obscure, and this trigger naming convention helps us know where to look for the problem.

T-SQL Trigger Notes

T-SQL does not support the *CREATE OR REPLACE* construct used in PL/SQL, though since MSSQL 2016 there is an equivalent CREATE OR ALTER. In earlier versions of MSSQL it is necessary to use *CREATE TRIGGER* the first time the trigger is created, and *ALTER TRIGGER* when making changes to an existing trigger.

To specify the update of a particular column in the AFTER clause in a T-SQL trigger, you specify just the function (INSERT/UPDATE) and then specify the column in the clause like so:

```
IF (UPDATE(column_name))
BEGIN
     (SQL lines)
END
```

We also use the clause *SET NOCOUNT ON*. This phrase prevents the display of the "Number of rows affected".

Notice that this is an AFTER trigger, which means that it runs after the insert or update of P_ONHAND. Most commercial DBMS support both BEFORE and AFTER triggers. Oracle and Microsoft SQL Server since MSSQL 2000 also implement INSTEAD OF triggers, which run instead of the triggering DML.

INSTEAD triggers are powerful. For example, you can use them to compute the value that actually should have been inserted in a row, or use them to prevent some users from updating particular columns.

Notice that this trigger updates every row in the PRODUCT table each time the trigger runs. While this is functionally correct it is extremely inefficient. This mistake is presented here to show you how easy it is to make mistakes like this. Such mistakes are common enough that triggers sometimes cause unsuspected database inefficiencies.

More Debugging of the Example

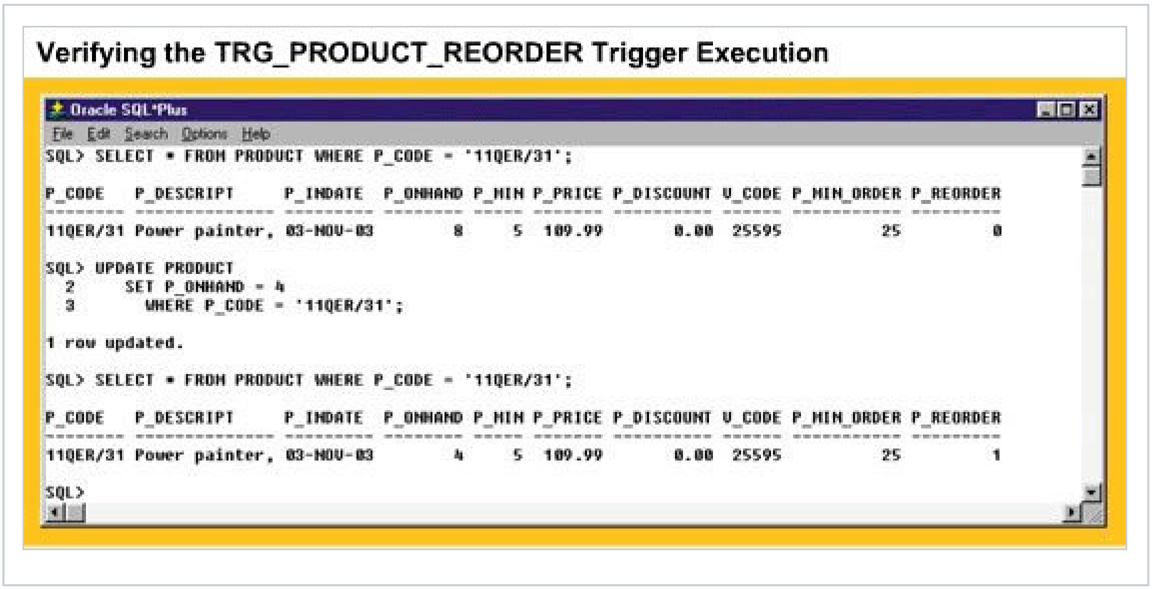
There is another serious design error in this trigger, other than the one that the authors point out in their debugging demonstration. The error is that this is a statement trigger rather than a FOR EACH ROW trigger. This trigger will run only once no matter how many rows are inserted or updated. This trigger is still functionally correct, because each update or insert recomputes all of the P_REORDER values, but this is still quite inefficient, and if we fixed the efficiency bug in the trigger so that it only updates the P_REORDER values for rows that are updated, this design bug would cause a correctness bug. To illustrate why this is important consider a SQL statement to update the PRODUCT table as part of the transaction that committed a shopping cart:

```
UPDATE product p
SET p.p_onhand = p.p_onhand -
(SELECT qty FROM SHOPPING_CART sc
WHERE sc.product_id = p.product_id);
```

The UPDATE statement above can update any number of rows of the product table. This is common.

I notice that there is probably a business rule error in this trigger. The trigger sets the reorder flag when p_onhand = p_min. Sometimes you want to have zero of an item in stock, for example for a special-order-only item. This trigger would make it awkward to maintain these inventories. For example, if p_min is zero, indicating that it is OK to have none in stock, then this trigger would set the reorder flag even if there were the intended zero in stock.

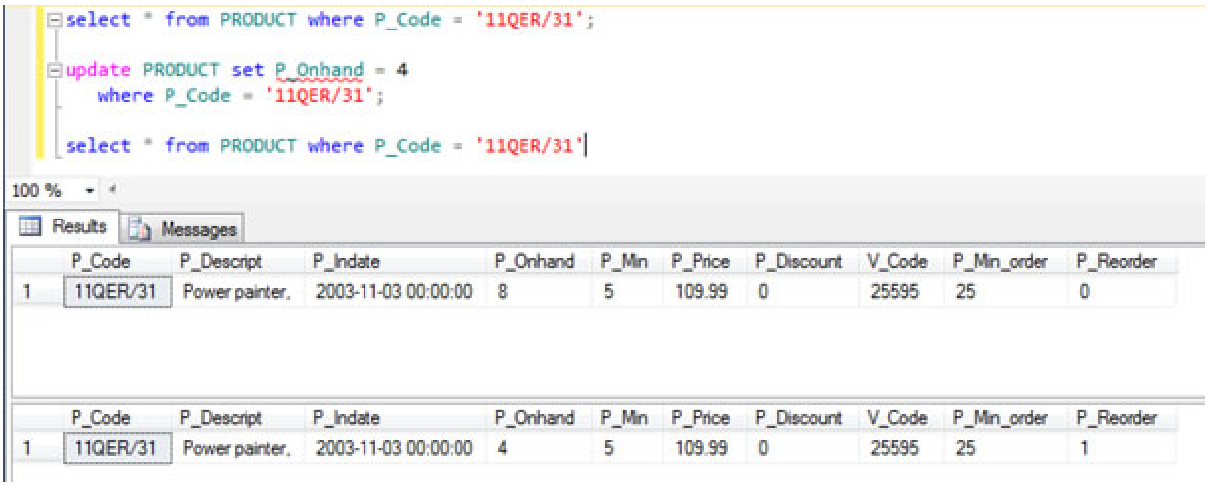
Next we follow the text with one of the tests of this trigger. Note that even a DBA can't invoke a trigger any other way than causing the triggering event, which in this case is a change to P_ONHAND.



Verifying the TRG_PRODUCT_REORDER Trigger Execution

```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter, 03-NOV-03         8     5  109.99       0.00  25595          25         0

SQL> UPDATE PRODUCT
  2     SET P_ONHAND = 4
  3       WHERE P_CODE = '11QER/31';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter, 03-NOV-03         4     5  109.99       0.00  25595          25         1

SQL>
```

Verifying the T-SQL Trigger



```
select * from PRODUCT where P_Code = '11QER/31';

update PRODUCT set P_Onhand = 4
    where P_Code = '11QER/31';

select * from PRODUCT where P_Code = '11QER/31'
```

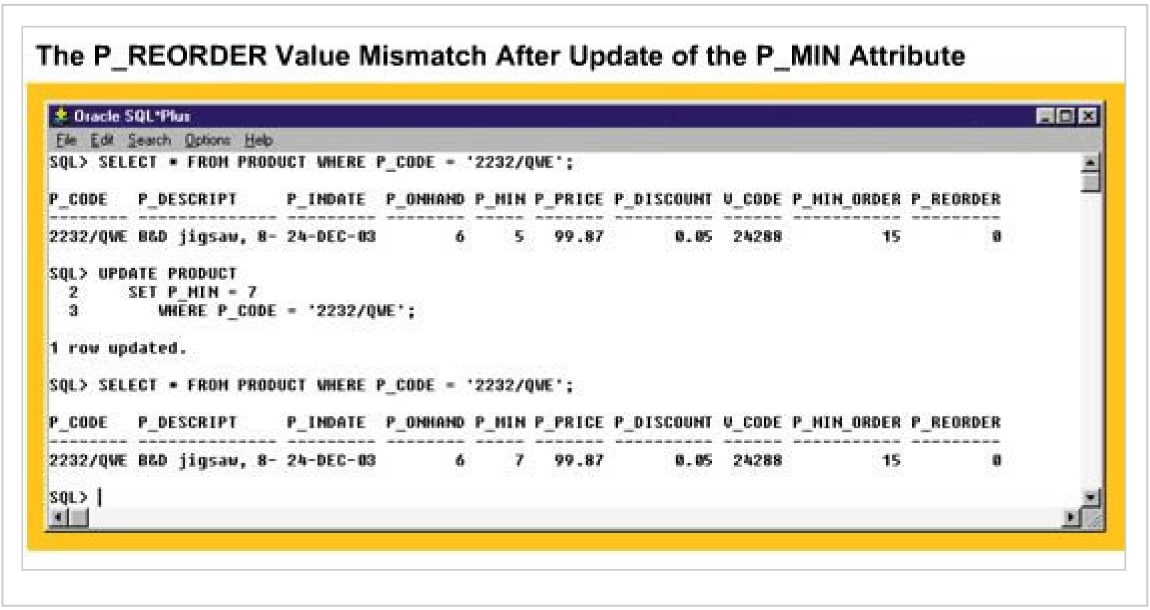
	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	11QER/31	Power painter,	2003-11-03 00:00:00	8	5	109.99	0	25595	25	0

	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	11QER/31	Power painter,	2003-11-03 00:00:00	4	5	109.99	0	25595	25	1

We see that the trigger is functionally correct for this update.

Further Debugging of the Example

The author's next test changes to the P_MIN column, and of course the trigger doesn't work correctly, because the trigger doesn't have updates of P_MIN as a trigger condition:

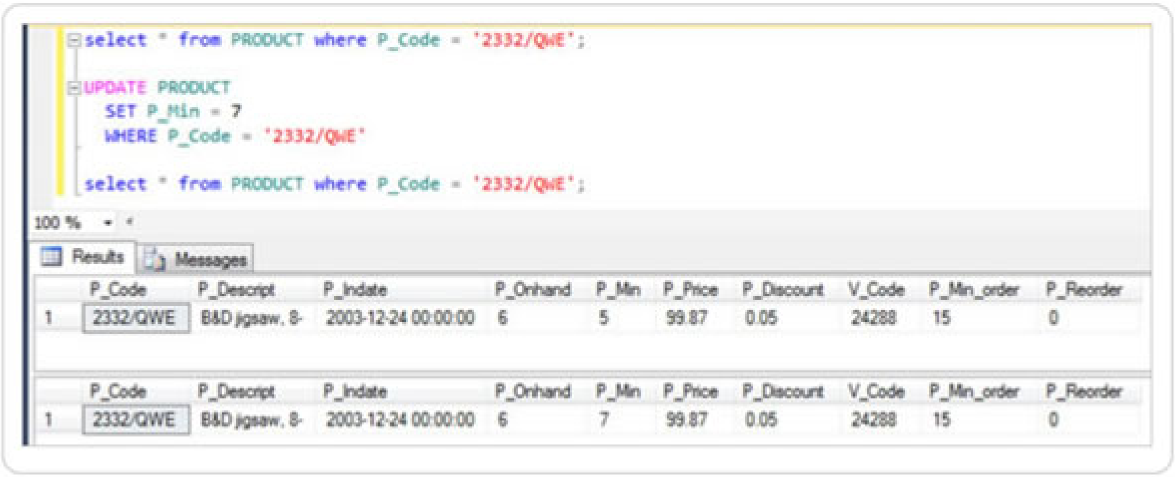

The P_REORDER Value Mismatch After Update of the P_MIN Attribute

```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '2232/QWE';

P_CODE   P_DESCRIPT       P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- ---------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
2232/QWE B&D jigsaw, 8- 24-DEC-03          6     5   99.87        0.05  24288          15         0

SQL> UPDATE PRODUCT
  2     SET P_MIN = 7
  3        WHERE P_CODE = '2232/QWE';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '2232/QWE';

P_CODE   P_DESCRIPT       P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- ---------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
2232/QWE B&D jigsaw, 8- 24-DEC-03          6     7   99.87        0.05  24288          15         0

SQL> |
```

Note that P_REORDER is still zero, even though P_ONHAND is less than P_MIN.

T-SQL Example

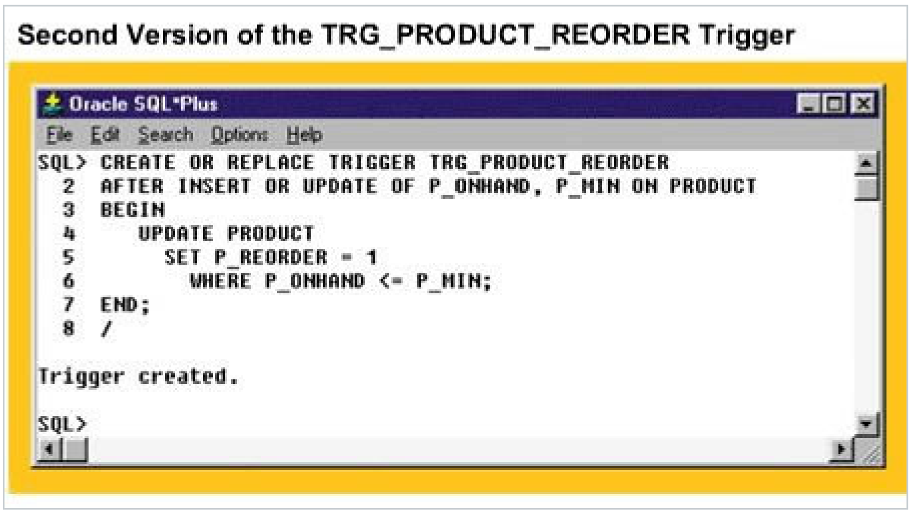
The P_REORDER Value Mismatch after Update of the P_MIN attribute


```
select * from PRODUCT where P_Code = '2332/QWE';

UPDATE PRODUCT
  SET P_Min = 7
  WHERE P_Code = '2332/QWE'

select * from PRODUCT where P_Code = '2332/QWE';
```

```
   P_Code    P_Descript      P_Indate              P_Onhand  P_Min  P_Price  P_Discount  V_Code  P_Min_order  P_Reorder
1  2332/QWE  B&D jigsaw, 8-  2003-12-24 00:00:00   6         5      99.87    0.05        24288   15           0

   P_Code    P_Descript      P_Indate              P_Onhand  P_Min  P_Price  P_Discount  V_Code  P_Min_order  P_Reorder
1  2332/QWE  B&D jigsaw, 8-  2003-12-24 00:00:00   6         7      99.87    0.05        24288   15           0
```

So we proceed to fix one of the bugs in the trigger:

We just add the clause (or UPDATE(P_MIN) to show that the trigger will be fired when P_MIN is updated.



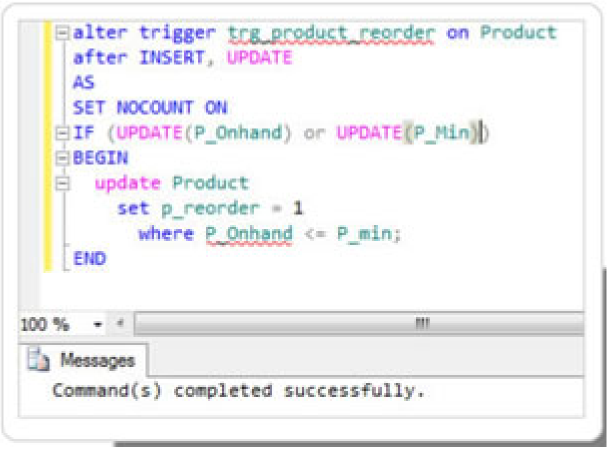
Second Version of the TRG_PRODUCT_REORDER Trigger

```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
  2  AFTER INSERT OR UPDATE OF P_ONHAND, P_MIN ON PRODUCT
  3  BEGIN
  4      UPDATE PRODUCT
  5         SET P_REORDER = 1
  6           WHERE P_ONHAND <= P_MIN;
  7  END;
  8  /

Trigger created.

SQL>
```

Note that this trigger still has three bugs—being a statement trigger when it should be a FOR EACH ROW trigger, updating every row in PRODUCT, and never setting P_REORDER to zero, even when P_ONHAND > P_MIN.

T-SQL Example

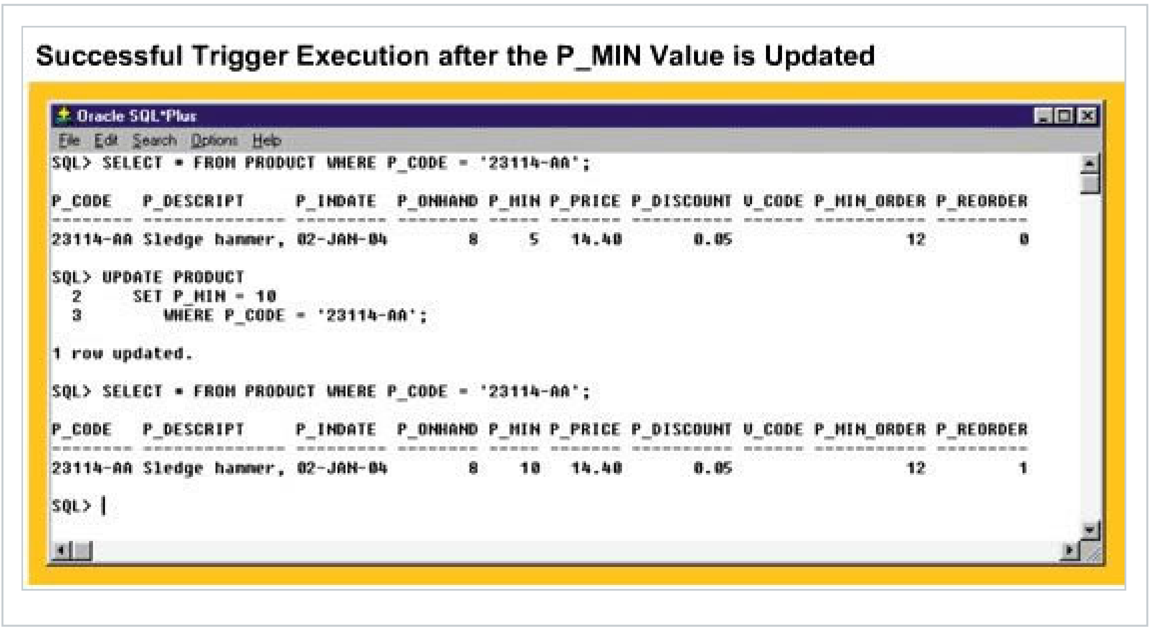
The second version of the TRG_PRODUCT_REORDER trigger:



```
alter trigger trg_product_reorder on Product
after INSERT, UPDATE
AS
SET NOCOUNT ON
IF (UPDATE(P_Onhand) or UPDATE(P_Min))
BEGIN
  update Product
    set p_reorder = 1
      where P_Onhand <= P_min;
END
```

```
Command(s) completed successfully.
```

We next test this trigger for increases in the value of P_MIN:

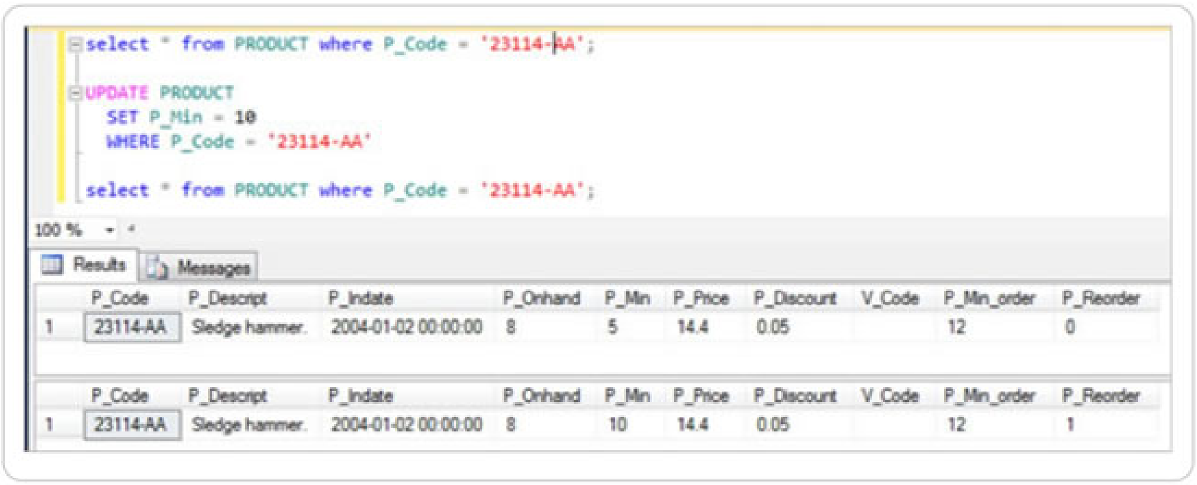


Successful Trigger Execution after the P_MIN Value is Updated

```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '23114-AA';

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
23114-AA Sledge hammer,  02-JAN-04        8     5   14.40       0.05                 12         0

SQL> UPDATE PRODUCT
  2     SET P_MIN = 10
  3        WHERE P_CODE = '23114-AA';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '23114-AA';

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
23114-AA Sledge hammer,  02-JAN-04        8    10   14.40       0.05                 12         1

SQL> |
```

T-SQL Example

The successful trigger execution:

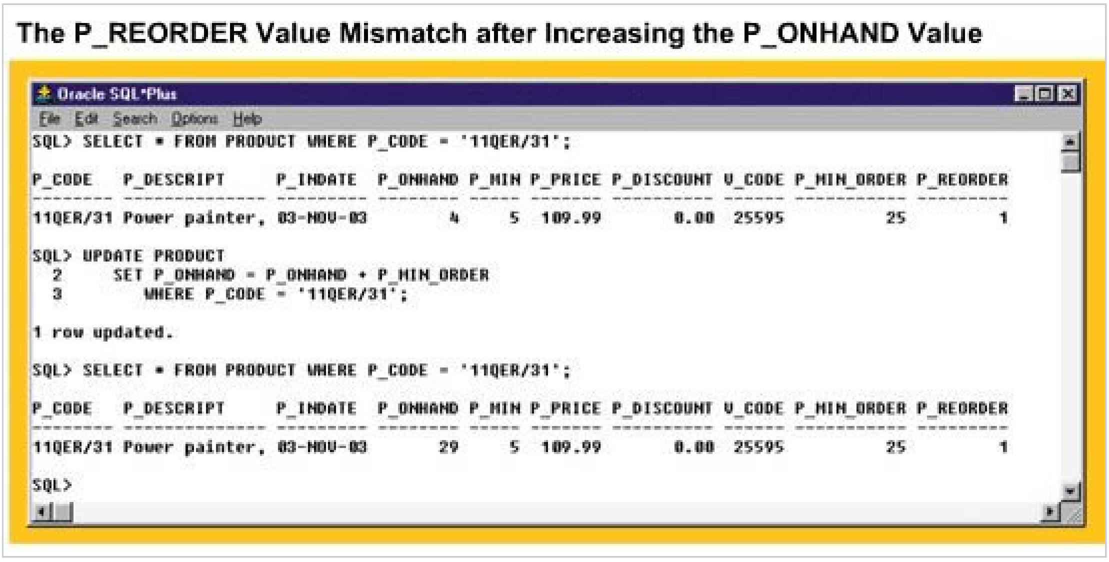


```
select * from PRODUCT where P_Code = '23114-AA';

UPDATE PRODUCT
  SET P_Min = 10
  WHERE P_Code = '23114-AA'

select * from PRODUCT where P_Code = '23114-AA';
```

	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	23114-AA	Sledge hammer.	2004-01-02 00:00:00	8	5	14.4	0.05		12	0

	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	23114-AA	Sledge hammer.	2004-01-02 00:00:00	8	10	14.4	0.05		12	1

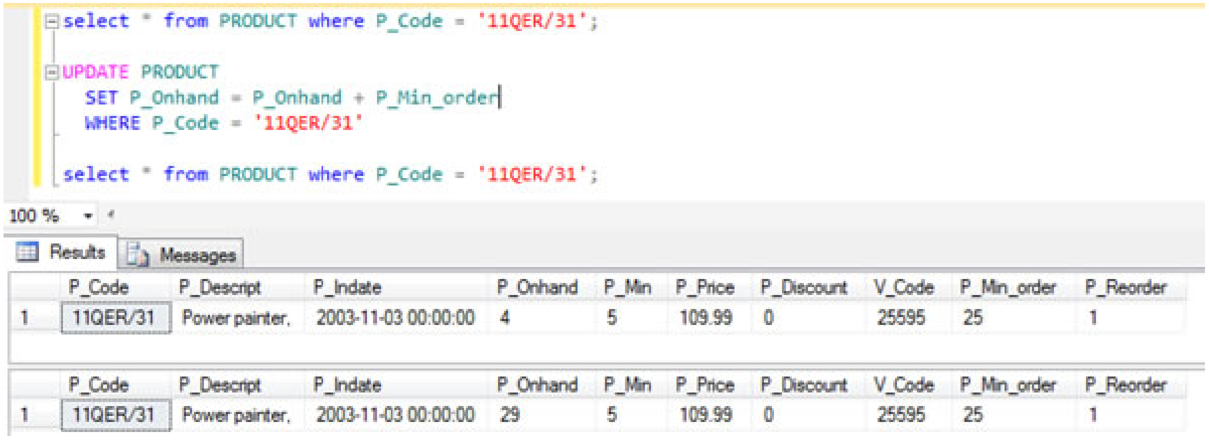
So now let's try increasing the value of P_ONHAND:


The P_REORDER Value Mismatch after Increasing the P_ONHAND Value

```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter,  03-NOV-03        4     5  109.99       0.00  25595          25         1

SQL> UPDATE PRODUCT
  2     SET P_ONHAND = P_ONHAND + P_MIN_ORDER
  3        WHERE P_CODE = '11QER/31';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE   P_DESCRIPT      P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- --------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter,  03-NOV-03       29     5  109.99       0.00  25595          25         1

SQL>
```

T-SQL Example

P_REORDER Value Mismatch after increasing the P_ONHAND Value:



```
select * from PRODUCT where P_Code = '11QER/31';

UPDATE PRODUCT
  SET P_Onhand = P_Onhand + P_Min_order
  WHERE P_Code = '11QER/31'

select * from PRODUCT where P_Code = '11QER/31';
```

	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	11QER/31	Power painter,	2003-11-03 00:00:00	4	5	109.99	0	25595	25	1

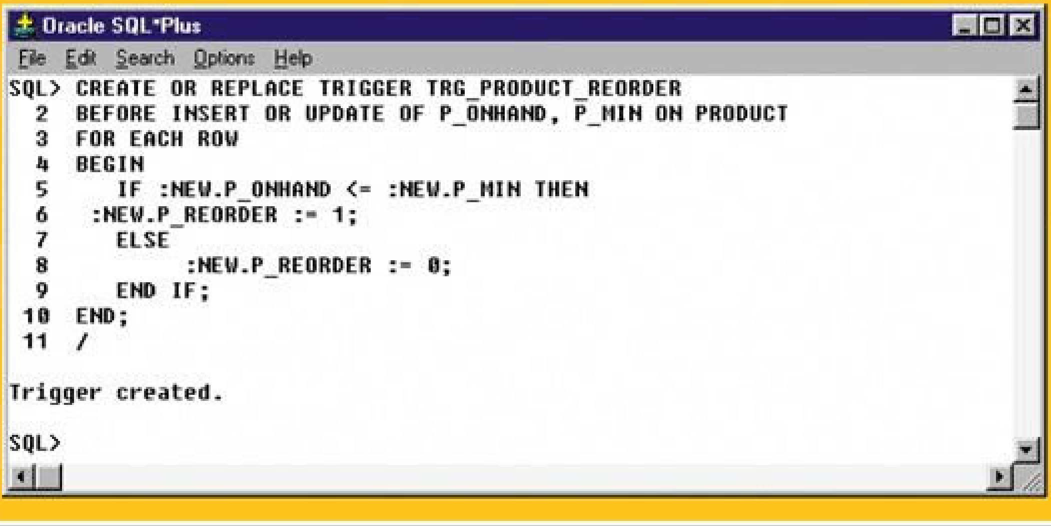
	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	11QER/31	Power painter,	2003-11-03 00:00:00	29	5	109.99	0	25595	25	1

Performance Tip

Responsibility-based design and development is the best practice for all database design and programming. The responsibility of this trigger is correctly maintaining the value of P_REORDER, no matter what happens. This localization of responsibility in the trigger is critical for maintainability, testability, correctness, and life cycle costs. For example, if there is any problem with P_REORDER there is only one place to check—this trigger.

So the trigger doesn't handle this correctly either, because P_REORDER = 1 when P_ONHAND is greater than P_MIN. This should not be surprising to programmers, because the trigger doesn't correctly take responsibility for the setting P_REORDER in all cases. The next example corrects all three of these errors:

The Third Version of the TRG_PRODUCT_REORDER Trigger



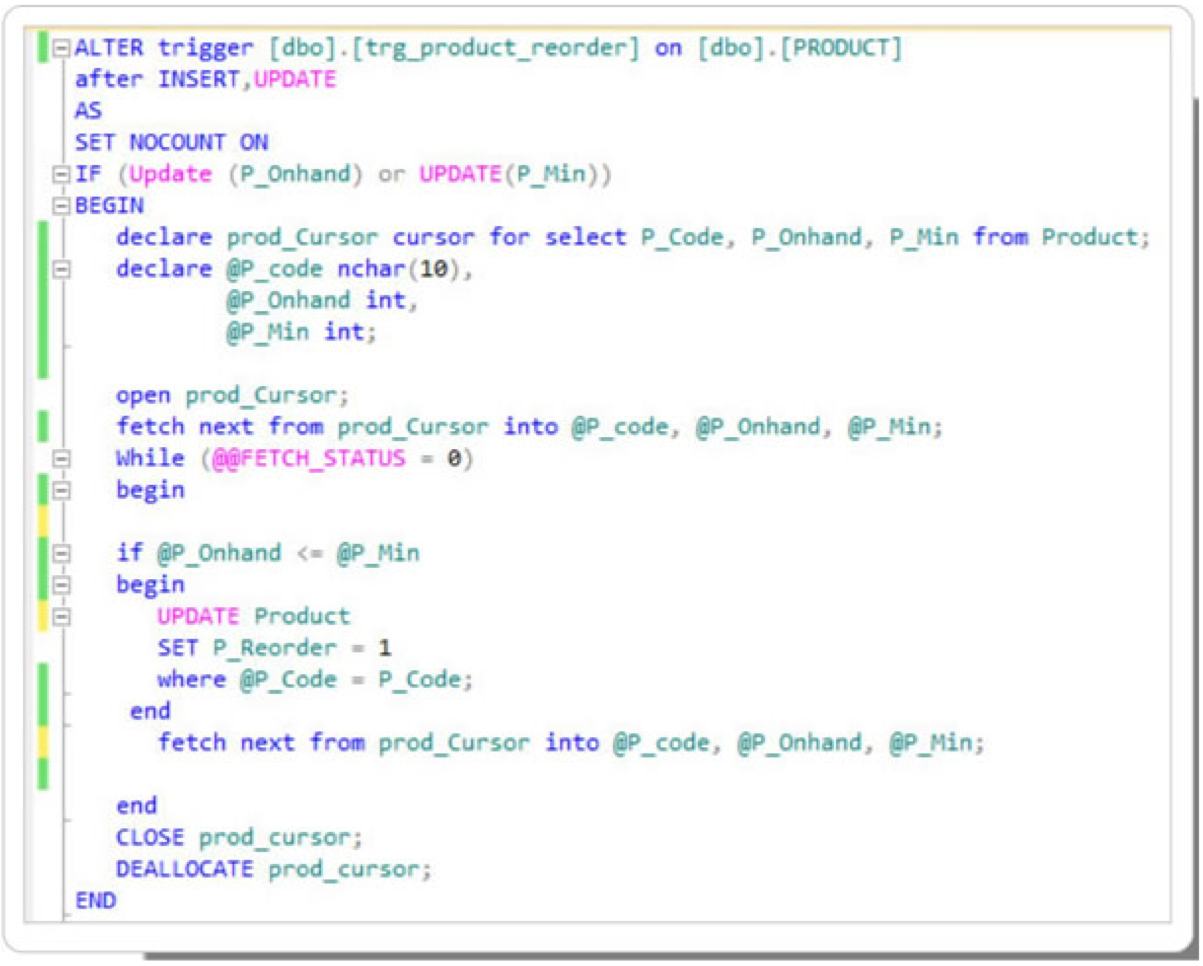
```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
  2  BEFORE INSERT OR UPDATE OF P_ONHAND, P_MIN ON PRODUCT
  3  FOR EACH ROW
  4  BEGIN
  5     IF :NEW.P_ONHAND <= :NEW.P_MIN THEN
  6   :NEW.P_REORDER := 1;
  7     ELSE
  8          :NEW.P_REORDER := 0;
  9     END IF;
 10  END;
 11  /

Trigger created.

SQL>
```

Note that it is a FOR EACH ROW trigger, as it should be.

This trigger will run for each row which is either new or for which either P_MIN or P_ONHAND is updated. We note that the *assertion* for which this trigger is responsible is that P_REORDER is zero when P_ONHAND is less than P_MIN and otherwise P_REORDER is one.

T-SQL Example

The third version of the TRG_PRODUCT_REORDER trigger:



```
ALTER trigger [dbo].[trg_product_reorder] on [dbo].[PRODUCT]
after INSERT,UPDATE
AS
SET NOCOUNT ON
IF (Update (P_Onhand) or UPDATE(P_Min))
BEGIN
    declare prod_Cursor cursor for select P_Code, P_Onhand, P_Min from Product;
    declare @P_code nchar(10),
            @P_Onhand int,
            @P_Min int;

    open prod_Cursor;
    fetch next from prod_Cursor into @P_code, @P_Onhand, @P_Min;
    While (@@FETCH_STATUS = 0)
    begin

    if @P_Onhand <= @P_Min
    begin
       UPDATE Product
       SET P_Reorder = 1
       where @P_Code = P_Code;
     end
       fetch next from prod_Cursor into @P_code, @P_Onhand, @P_Min;

    end
    CLOSE prod_cursor;
    DEALLOCATE prod_cursor;
END
```

Note that this uses a *cursor*, which is a subset of data.

Like the other versions of this trigger, we start by checking to see if the P_Onhand or P_Min fields have been updated. If so, we want to check every record in the Product table to see if any other values for P_Reorder need to be adjusted. So we start by declaring a cursor. What we're going to "put into" the cursor is the set of values determined by the select statement, in this case all the product codes, P_Onhand values and P_Min values from the entire Product table. Temporary variables that will "hold" the record values are also declared in this section.

The "Fetch Next" statement loops through every record selected into the cursor and only adjusts those for which the onhand value is less than the minimum value. FETCH NEXT …INTO sets the temporary variables equal to the record values for each record it reads. We set up a "WHILE" loop to loop through all the records in the cursor. If all the records in the cursor have been processed, the variable @@FETCH_STATUS is no longer zero and there is no need to continue processing the data. Note, also, that it's important to fetch the next record right before the end of the WHILE loop, so the trigger doesn't continue to process the same record over and over again.

This trigger will run for each row which is either new or for which either P_MIN or P_ONHAND is updated. We note that the *assertion* for which this trigger is responsible is that P_REORDER is zero when P_ONHAND is less than P_MIN and otherwise P_REORDER is one.

Definition of Assertion

An *assertion* is a Boolean predicate which should always be true. Assertions are important in programming, and there is support for assertions in programming languages. Some optimizing compilers will try to prove that assertions are true, and if they can do so will eliminate runtime tests for the assertions. Compilers may also use the fact that assertions are true in optimizing the code. If you wanted to check this assertion you could add a CHECK constraint, and the check constraint would run after this trigger. In practice you would probably not do this in a busy production database, because of the overhead.

Test Yourself

What are some of the reasons to use a FOR EACH ROW or statenebt-level trigger?

FOR EACH ROW triggers execute each time a new row is updated or inserted, which is a good way to maintain the data.

This is true.

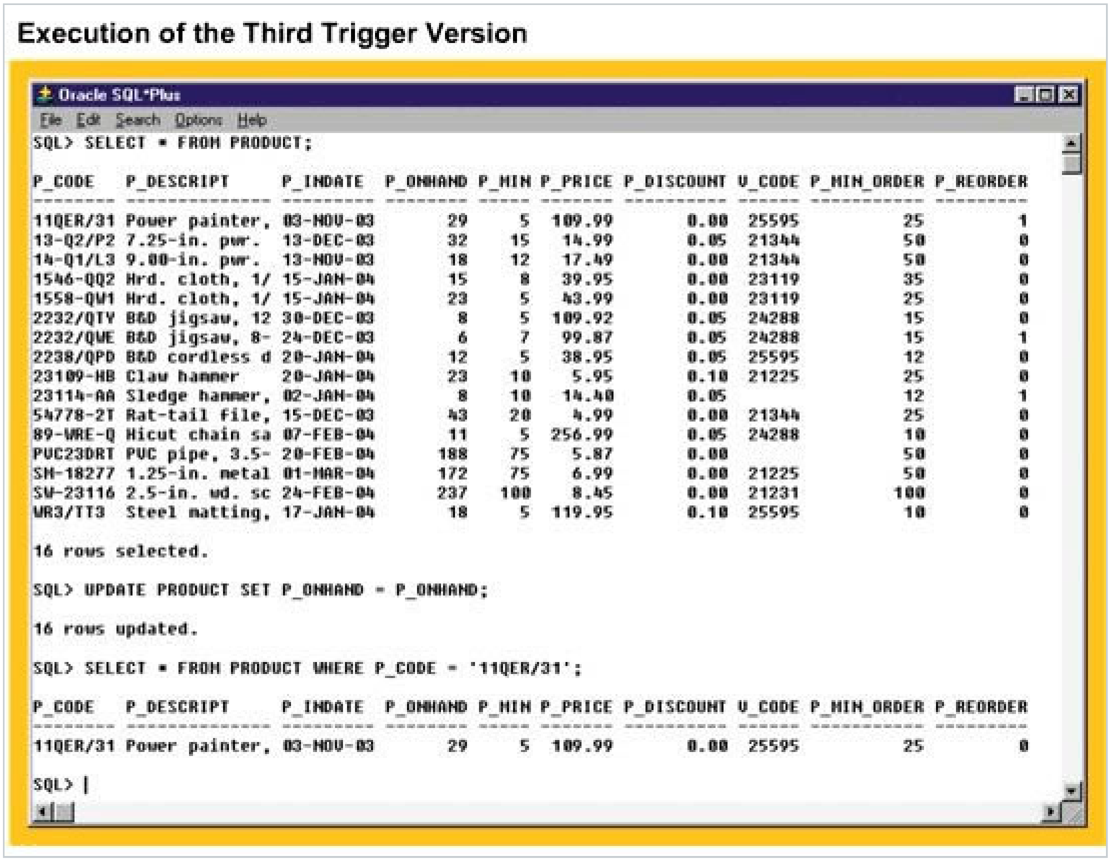
A FOR EACH ROW trigger will only run once for an UPDATE statement.

This is false. A FOR EACH ROW trigger will run once for each row which is updated when the update satisfies the conditions of the trigger.

A FOR EACH ROW trigger only executes once, but it updates the entire table.

This is false. A FOR EACH ROW trigger runs once for each updated row in the table that satisfies the conditions of the trigger.

We next test this trigger, by updating every row in the PRODUCT table, but not changing any values:

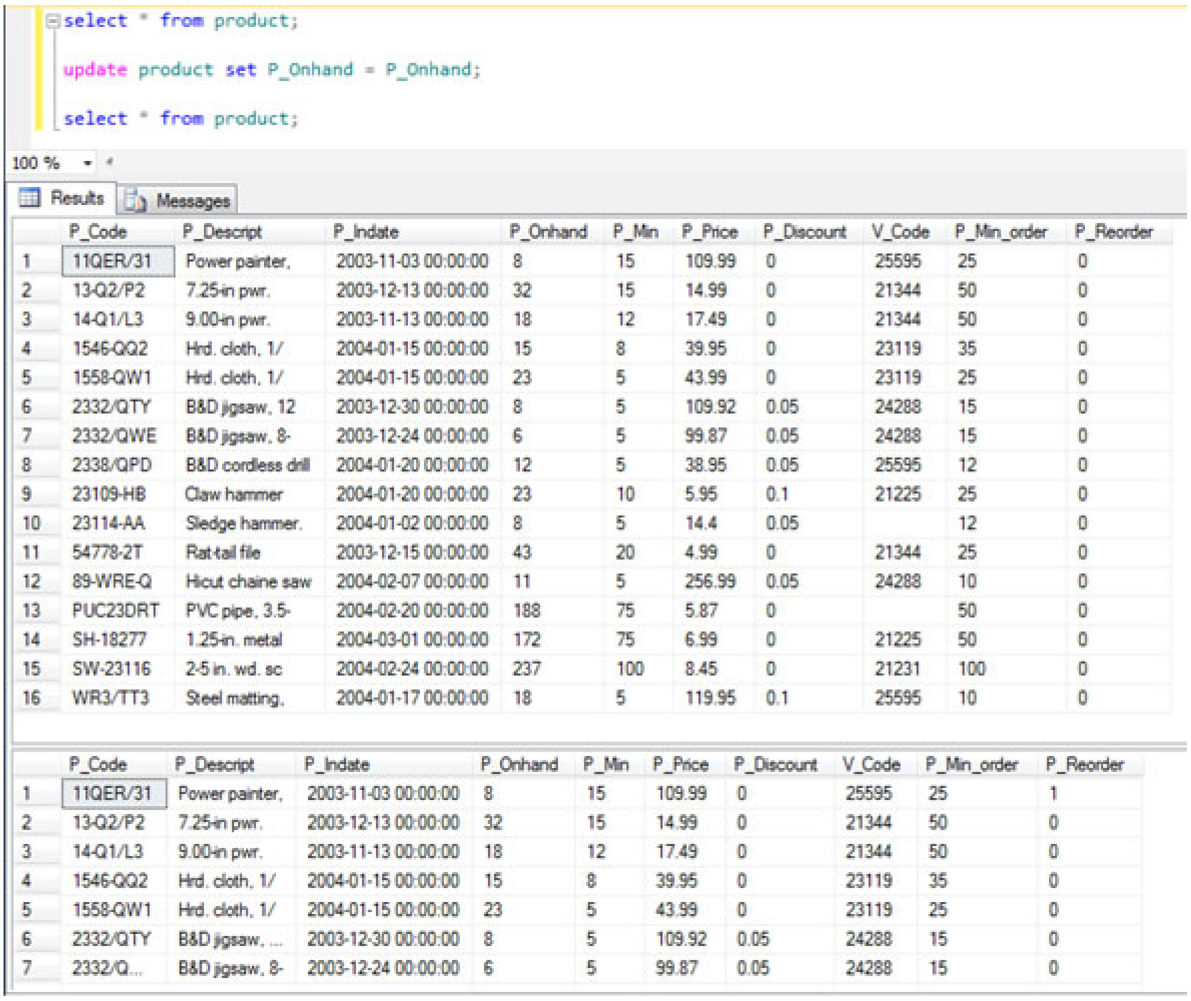


Execution of the Third Trigger Version

```
SQL> SELECT * FROM PRODUCT;

P_CODE   P_DESCRIPT     P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- -------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter, 03-NOV-03       29     5  109.99       0.00  25595          25         1
13-Q2/P2 7.25-in. pwr.  13-DEC-03       32    15   14.99       0.05  21344          50         0
14-Q1/L3 9.00-in. pwr.  13-NOV-03       18    12   17.49       0.00  21344          50         0
1546-QQ2 Hrd. cloth, 1/ 15-JAN-04       15     8   39.95       0.00  23119          35         0
1558-QW1 Hrd. cloth, 1/ 15-JAN-04       23     5   43.99       0.00  23119          25         0
2232/QTY B&D jigsaw, 12 30-DEC-03        8     5  109.92       0.05  24288          15         0
2232/QWE B&D jigsaw, 8- 24-DEC-03        6     7   99.87       0.05  24288          15         1
2238/QPD B&D cordless d 20-JAN-04       12     5   38.95       0.05  25595          12         0
23109-HB Claw hammer    20-JAN-04       23    10    5.95       0.10  21225          25         0
23114-AA Sledge hammer, 02-JAN-04        8    10   14.40       0.05                 12         1
54778-2T Rat-tail file, 15-DEC-03       43    20    4.99       0.00  21344          25         0
89-WRE-Q Hicut chain sa 07-FEB-04       11     5  256.99       0.05  24288          10         0
PVC23DRT PVC pipe, 3.5- 20-FEB-04      188    75    5.87       0.00                 50         0
SM-18277 1.25-in. metal 01-MAR-04      172    75    6.99       0.00  21225          50         0
SW-23116 2.5-in. wd. sc 24-FEB-04      237   100    8.45       0.00  21231         100         0
WR3/TT3  Steel matting, 17-JAN-04       18     5  119.95       0.10  25595          10         0

16 rows selected.

SQL> UPDATE PRODUCT SET P_ONHAND = P_ONHAND;

16 rows updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE   P_DESCRIPT     P_INDATE  P_ONHAND P_MIN P_PRICE P_DISCOUNT V_CODE P_MIN_ORDER P_REORDER
-------- -------------- --------- -------- ----- ------- ---------- ------ ----------- ---------
11QER/31 Power painter, 03-NOV-03       29     5  109.99       0.00  25595          25         0

SQL> |
```

T-SQL Example

Execution of the Third Trigger Version:



```
select * from product;

update product set P_Onhand = P_Onhand;

select * from product;
```

100 %

Results | Messages

	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	11QER/31	Power painter,	2003-11-03 00:00:00	8	15	109.99	0	25595	25	0
2	13-Q2/P2	7.25-in pwr.	2003-12-13 00:00:00	32	15	14.99	0	21344	50	0
3	14-Q1/L3	9.00-in pwr.	2003-11-13 00:00:00	18	12	17.49	0	21344	50	0
4	1546-QQ2	Hrd. cloth, 1/	2004-01-15 00:00:00	15	8	39.95	0	23119	35	0
5	1558-QW1	Hrd. cloth, 1/	2004-01-15 00:00:00	23	5	43.99	0	23119	25	0
6	2232/QTY	B&D jigsaw, 12	2003-12-30 00:00:00	8	5	109.92	0.05	24288	15	0
7	2232/QWE	B&D jigsaw, 8-	2003-12-24 00:00:00	6	5	99.87	0.05	24288	15	0
8	2238/QPD	B&D cordless drill	2004-01-20 00:00:00	12	5	38.95	0.05	25595	12	0
9	23109-HB	Claw hammer	2004-01-20 00:00:00	23	10	5.95	0.1	21225	25	0
10	23114-AA	Sledge hammer,	2004-01-02 00:00:00	8	5	14.4	0.05		12	0
11	54778-2T	Rat-tail file	2003-12-15 00:00:00	43	20	4.99	0	21344	25	0
12	89-WRE-Q	Hicut chaine saw	2004-02-07 00:00:00	11	5	256.99	0.05	24288	10	0
13	PVC23DRT	PVC pipe, 3.5-	2004-02-20 00:00:00	188	75	5.87	0		50	0
14	SH-18277	1.25-in. metal	2004-03-01 00:00:00	172	75	6.99	0	21225	50	0
15	SW-23116	2-5 in. wd. sc	2004-02-24 00:00:00	237	100	8.45	0	21231	100	0
16	WR3/TT3	Steel matting,	2004-01-17 00:00:00	18	5	119.95	0.1	25595	10	0

	P_Code	P_Descript	P_Indate	P_Onhand	P_Min	P_Price	P_Discount	V_Code	P_Min_order	P_Reorder
1	11QER/31	Power painter,	2003-11-03 00:00:00	8	15	109.99	0	25595	25	1
2	13-Q2/P2	7.25-in pwr.	2003-12-13 00:00:00	32	15	14.99	0	21344	50	0
3	14-Q1/L3	9.00-in pwr.	2003-11-13 00:00:00	18	12	17.49	0	21344	50	0
4	1546-QQ2	Hrd. cloth, 1/	2004-01-15 00:00:00	15	8	39.95	0	23119	35	0
5	1558-QW1	Hrd. cloth, 1/	2004-01-15 00:00:00	23	5	43.99	0	23119	25	0
6	2232/QTY	B&D jigsaw, ...	2003-12-30 00:00:00	8	5	109.92	0.05	24288	15	0
7	2232/Q...	B&D jigsaw, 8-	2003-12-24 00:00:00	6	5	99.87	0.05	24288	15	0

Notice that the first row in the PRODUCT table has an incorrect value for P_REORDER, and that this error is corrected after running the trigger on all rows. Note that the trigger would run in exactly the same way on all rows if we had said:

```
UPDATE product SET p_min = p_min;
```

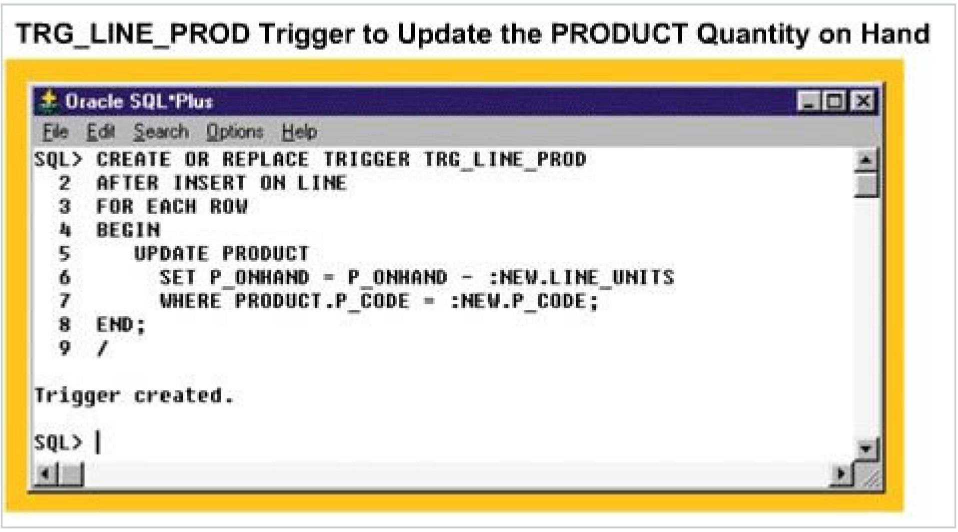
This appears to be an industrial-strength trigger that can bear full responsibility for the correctness of P_REORDER.

Testing Tip

Triggers are difficult to test, so I usually develop a procedure to test all of the cases, which in this case would include an insert, and at least three updates each of P_ONHAND and P_MIN, where one update increased and decreased each value and one update left it unchanged. The test procedure would normally insert test data in the PRODUCT table, run the tests, and then delete the test data. This can be done most safely against a production database by not committing the test data inserts and then rolling back the transaction that inserts the test data when the tests have been completed. Thus if the test crashes for any reason, the inserts will be automatically rolled back. I usually do such tests within PL/SQL or T-SQL because then I can implement good exception handling, which we will cover later in the lecture.

Decrementing Triggers

We next continue with the presentation based on the example in the book, with a “toy” trigger that decrements product.p_onhand when a new row is inserted in the LINE table.

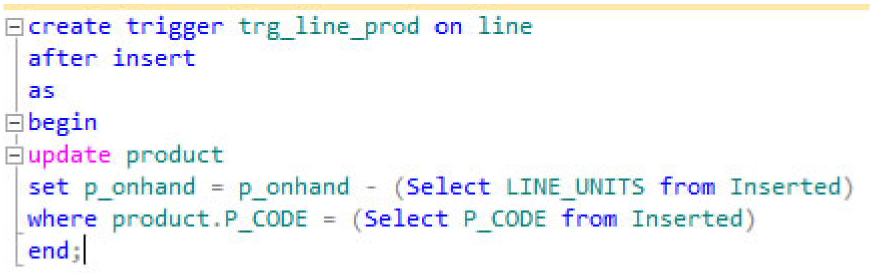


TRG_LINE_PROD Trigger to Update the PRODUCT Quantity on Hand

```
SQL> CREATE OR REPLACE TRIGGER TRG_LINE_PROD
  2  AFTER INSERT ON LINE
  3  FOR EACH ROW
  4  BEGIN
  5     UPDATE PRODUCT
  6       SET P_ONHAND = P_ONHAND - :NEW.LINE_UNITS
  7       WHERE PRODUCT.P_CODE = :NEW.P_CODE;
  8  END;
  9  /

Trigger created.

SQL> |
```

T-SQL Example

TRG_LINE_PROD trigger to update the ONHAND quantity:



```
create trigger trg_line_prod on line
after insert
as
begin
update product
set p_onhand = p_onhand - (Select LINE_UNITS from Inserted)
where product.P_CODE = (Select P_CODE from Inserted)
end;
```

I termed this a “toy” trigger because it is not a good design to implement this functionality in this trigger. The main reasons that this is not a good design are:

- The decrementing of the product.p_onhand column should happen when the goods are picked and shipped, and not when the line items are added. It is a common business practice to accept orders for goods and then ship them when they are received. This is part of the basis for the efficiency improvements from tight supply chain integration.
- This trigger would cause problems when rows are inserted into the line table for back orders, reorders, and other reasons that should not change product.p_onhand.
- This trigger changes p_onhand, yet the trigger is not competent to take responsibility for p_onhand, which can also change when goods are received, shipped, or inventoried. This is hence a rather complex responsibility that cannot be localized in this trigger. A better approach would be to give responsibility for product.p_onhand to a PL/SQL package responsible for the warehouse, shipping and receiving functions.

- This trigger does not even attempt to handle the case where the line table is updated.

Being Picked

Picking refers to the operations in warehouses where goods for a shipping order are removed from stock.

Test Yourself

When would you use the construct :OLD in a trigger? (Check all that are true.)

You would use :OLD when you want to insert the unchanged data into an Archive table.

This is true.

You would use :OLD when you want to update old data before the row is updated.

This is false. You will not be able to use ":OLD" to change data that is in the process of changing.

You would use :OLD when you want the trigger to access the data values in a record before they may have been changed.

This is true. This is the basic function of :OLD.

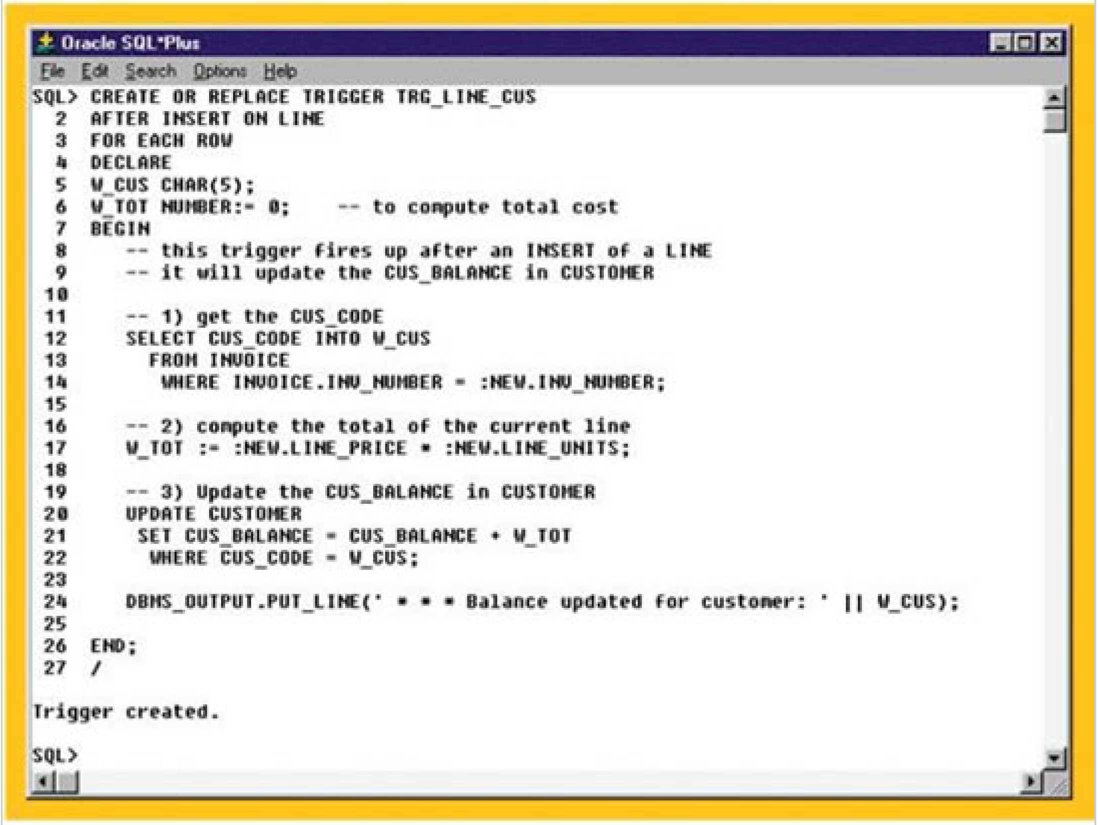
You can change the values in the database by changing the values in :OLD.

This is false. Changes to :OLD do not change the database. Changes to :NEW do.

Triggers That Update

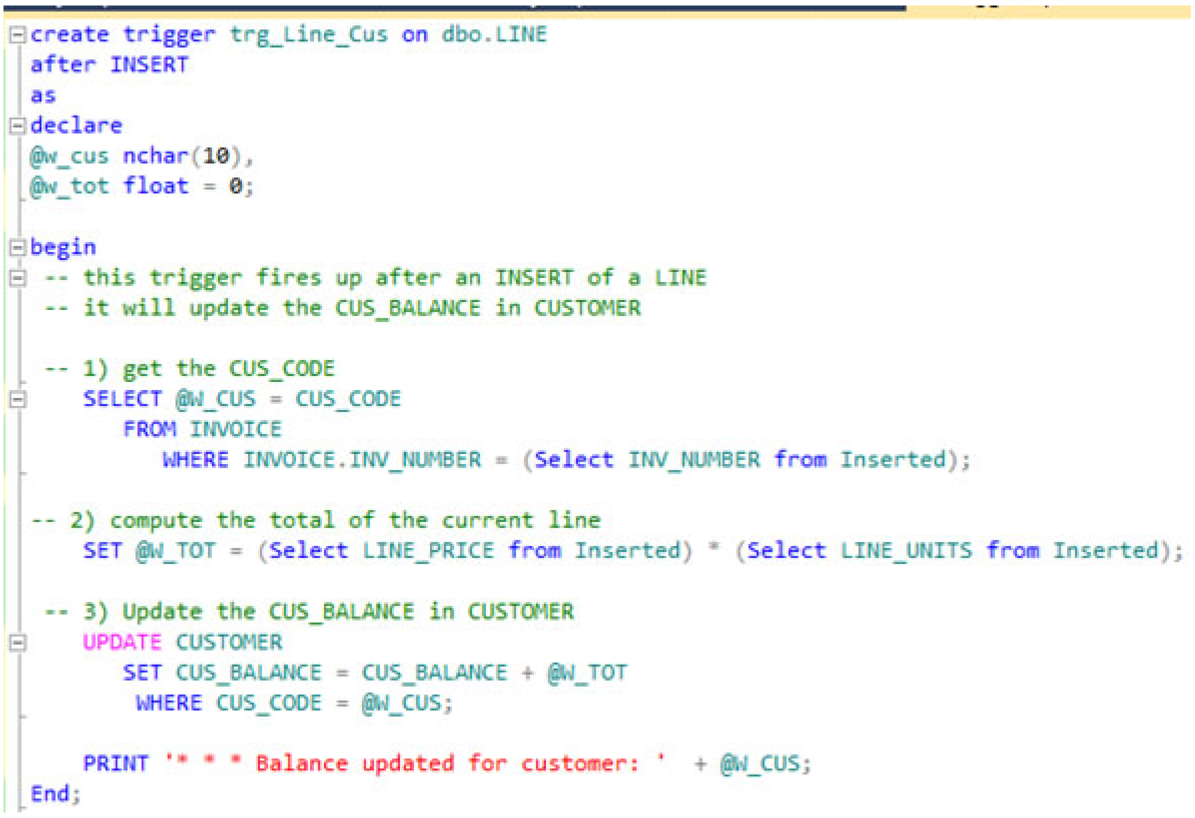
We now continue with the examples in the text, this time with the trigger that updates the customer.cus.balance when a row is inserted into the line table. As you may guess this is not a good design, but this makes it useful for teaching both good design and some additional features of triggers.

TRG_LINE_CUS Trigger to Update the Customer Balance



```
SQL> CREATE OR REPLACE TRIGGER TRG_LINE_CUS
  2  AFTER INSERT ON LINE
  3  FOR EACH ROW
  4  DECLARE
  5  W_CUS CHAR(5);
  6  W_TOT NUMBER:= 0;     -- to compute total cost
  7  BEGIN
  8     -- this trigger fires up after an INSERT of a LINE
  9     -- it will update the CUS_BALANCE in CUSTOMER
 10
 11     -- 1) get the CUS_CODE
 12     SELECT CUS_CODE INTO W_CUS
 13       FROM INVOICE
 14        WHERE INVOICE.INV_NUMBER = :NEW.INV_NUMBER;
 15
 16     -- 2) compute the total of the current line
 17     W_TOT := :NEW.LINE_PRICE * :NEW.LINE_UNITS;
 18
 19     -- 3) Update the CUS_BALANCE in CUSTOMER
 20     UPDATE CUSTOMER
 21      SET CUS_BALANCE = CUS_BALANCE + W_TOT
 22       WHERE CUS_CODE = W_CUS;
 23
 24     DBMS_OUTPUT.PUT_LINE(' * * * Balance updated for customer: ' || W_CUS);
 25
 26  END;
 27  /

Trigger created.

SQL>
```

T-SQL Version of TRG_LINE_CUS Trigger



```
create trigger trg_Line_Cus on dbo.LINE
after INSERT
as
declare
@w_cus nchar(10),
@w_tot float = 0;

begin
 -- this trigger fires up after an INSERT of a LINE
 -- it will update the CUS_BALANCE in CUSTOMER

 -- 1) get the CUS_CODE
    SELECT @W_CUS = CUS_CODE
       FROM INVOICE
          WHERE INVOICE.INV_NUMBER = (Select INV_NUMBER from Inserted);

-- 2) compute the total of the current line
    SET @W_TOT = (Select LINE_PRICE from Inserted) * (Select LINE_UNITS from Inserted);

 -- 3) Update the CUS_BALANCE in CUSTOMER
    UPDATE CUSTOMER
       SET CUS_BALANCE = CUS_BALANCE + @W_TOT
        WHERE CUS_CODE = @W_CUS;

    PRINT '* * * Balance updated for customer: '  + @W_CUS;
End;
```

Note that the comment *scan stopper* symbol in PL/SQL is two adjacent hyphens.

Definition of a Scan Stopper

A *scan stopper* is a symbol that causes the scanner, which is the first stage of parsing of a program, to stop looking for tokens, but to skip to the end of the current line. The scan stopper symbol in C++ and Java is two adjacent forward slashes.

Notice that this trigger has a declarations section between DECLARE and BEGIN, just as in the PL/SQL blocks that begin with DECLARE.

This trigger doesn't handle the case where the line table is updated, so it is not of industrial strength.

Notice also that the SELECT statement beginning on line 12 does not have any checking for whether a CUS_CODE was actually selected into W_CUS. If the invoice number field of the newly inserted record is not in the invoice table, then W_CUS will be left NULL, and the UPDATE statement that begins on line 20 would do nothing, and the trigger wouldn't give any hint that there was an error. The customer would just not have their cus_balance increased. This is a correctness error in this trigger.

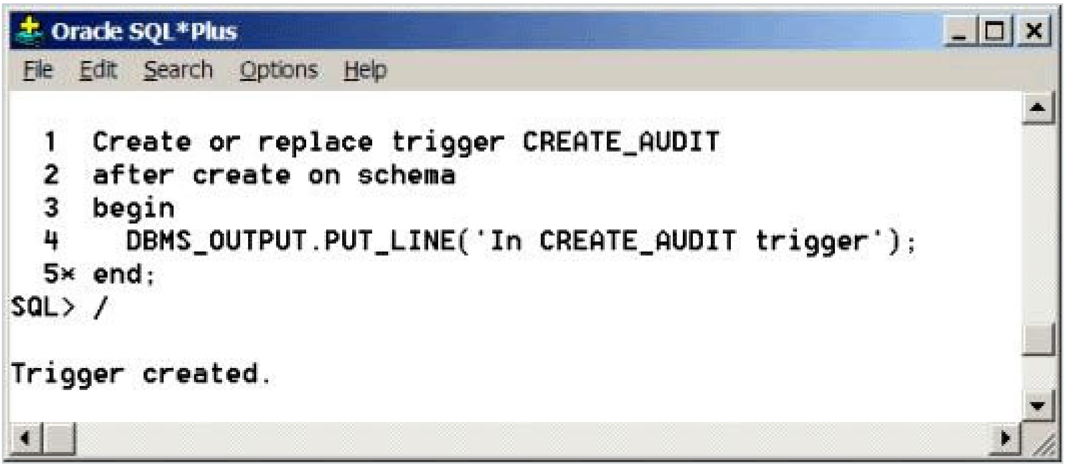
Design Tip

Triggers, stored procedures, and other database objects should take full responsibility for the correctness of what they are attempting to accomplish (their responsibilities) and generate an error if any of their inputs or preconditions is violated.

Note that this trigger updates the customer balance for each row inserted into the line table. This is inefficient, apart from the correctness problems mentioned above. A better design would localize this functionality in a stored procedure that implements the committing of an order, and this stored procedure would add up the amounts for the line items, and add this total invoice amount to the customer's balance in one update statement.

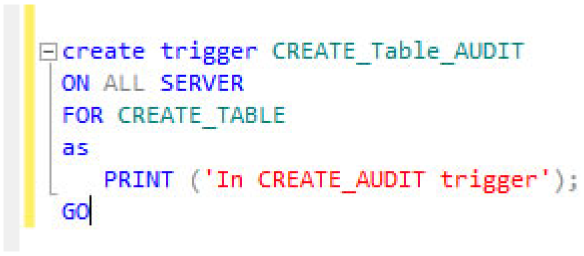
DDL Triggers

DDL triggers are database code that is associated with DDL operations such as the creation of a database object. Consider the following example:



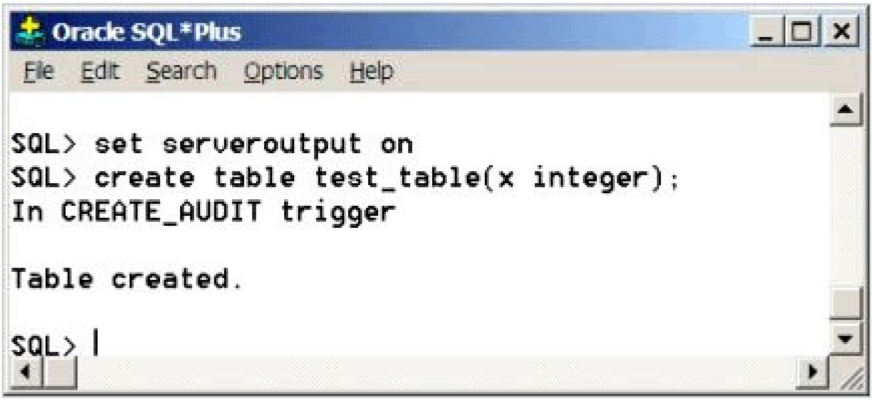
```
  1  Create or replace trigger CREATE_AUDIT
  2  after create on schema
  3  begin
  4    DBMS_OUTPUT.PUT_LINE('In CREATE_AUDIT trigger');
  5* end;
SQL> /

Trigger created.
```

The CREATE_AUDIT Trigger in T-SQL



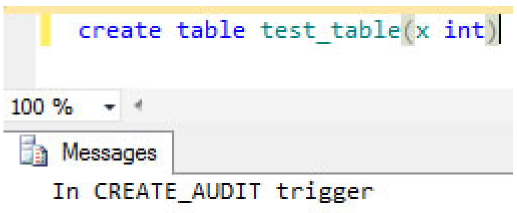
```
create trigger CREATE_Table_AUDIT
ON ALL SERVER
FOR CREATE_TABLE
as
   PRINT ('In CREATE_AUDIT trigger');
GO
```

We now test this trigger by creating a table:



```
SQL> set serveroutput on
SQL> create table test_table(x integer);
In CREATE_AUDIT trigger

Table created.

SQL> |
```

Test_table Created in T-SQL



```
create table test_table(x int)
```

```
In CREATE_AUDIT trigger
```

In a production application the DDL trigger would have logged the create operation, and may have alerted the DBA via email. The ROLLBACK command can be used in trigger to prevent DDL operations as in the following example:

A few words are appropriate about the call to the built-in function RAISE_APPLICATION_ERROR. The first argument is a binary_integer error code number, which must be in the range -20000 to -20999. Believe it or not, the first 20,000 negative error codes are used by Oracle for its own exceptions, and even the exception numbers from -20000 to -20005 are used by packages such as DBMS_OUTPUT and DBMS_DESCRIBE, so to be on the safe side it is best to use error code numbers between -20006 and -20999. The second argument is the error message, which you saw above when we tried to create the table table2. Note that the error number that Oracle sent was ORA-20999, which is the error number that we specified in the call to RAISE_APPLICATION_ERROR. The third argument to RAISE_APPLICATION_ERROR is a Boolean which tells Oracle whether to keep the errors in the error stack or replace existing errors; we told Oracle to replace existing errors on the stack. So now you get an idea of how Oracle generates all those error messages. This before DDL trigger prevented the execution of the create table statement, and if you checked you would see that table 2 doesn't exist.

T-SQL Version of Stop-DDL Trigger

Just a note here. You probably noticed that you can see the effects of the previous trigger as well. Since this trigger has a different name, they both act on the create table command. Debugging is sometimes difficult if more than one trigger is running on a table. In SQL Server, you can find out which server triggers exist with the command:

```
SELECT * FROM sys.server_triggers
```

Startup and Logon Triggers

Startup Triggers

Oracle supports triggers that run when the instance is started; consider:

```
Create or replace trigger START_TRG after startup on database
begin
/* do whatever */ end;
```

The code between *begin* and *end* will run each time the database instance is started. This is useful for operations such as setting up links to other databases.

Logon Triggers

SQL Server and Oracle also lets us create triggers that run whenever a user attempts to logon to the database. These triggers fire after the user has been authenticated, but before the session starts. The triggers do not fire if the user fails authentication.

I kept my SQLPlus session running, and created this trigger from TOAD for a reason that will be obvious shortly.

This innocent looking trigger just counts the number of user tables and then divides that number by zero, which raises an unhandled exception. By the ANSI standards this rolls back the transaction that triggered this event. So let's go back to our SQLPlus session and test this:

I have created what could be a serious problem. Not only can I no longer login to my database instance using any user name (even SYS as SYSDBA doesn't have enough privilege to override this trigger), but my SQLPlus connection was terminated by the error! Fortunately I anticipated this, and still have the TOAD connection that I used to create the logon trigger.

So I dropped this dangerous logon trigger; what a relief! If I hadn't been able to drop it I would have been stuck and would have had to reinstall Oracle to create any new connections to the database.

Some final remarks about triggers before we move on to stored procedures. The first is that you can enable or disable triggers, using this syntax:

```
Alter trigger <trigger name> enable; Alter trigger <trigger name>
disable;
```

To disable or enable a trigger you must be the owner of the trigger or have ALTER ANY TRIGGER privilege, which DBAs have. A DBA would normally disable a DDL blocking trigger before making a DDL change to a database, make the DDL changes, and then re-enable the DDL trigger before putting the database back in production.

T-SQL Example

```
enable trigger <trigger name> ON DATABASE(or ON ALL SERVER);
disable trigger <trigger name> on <schema>.<tableName>;
```

To disable or enable a trigger for one table, you must have ALTER privilege on the table.

To disable or enable a trigger ON ALL SERVER, you must have CONTROL SERVER permission on the server, which is common at the DBA level. To enable a trigger ON DATABASE, you would need ALTER ANY DATABASE permission.

Disabling a trigger is not the same as dropping the trigger. When you disable a trigger, it still exists in the database but will not fire (Microsoft, 2013).

Logon triggers can be used to track logon activity or to restrict logins. It is possible to create a logon trigger that prevents *all* users from logging on, so they must be used with care.

Test Yourself

Which of the following are true of triggers? (Check all that are true.)

The owner of a trigger can enable and disable it.

This is true.

When a trigger is disabled it will not run even if the triggering event occurs.

This is true.

A trigger fires based on an event. The only way to prevent a trigger from firing is to disable or drop the trigger *before* the event occurs.

This is true.

Stored Procedures

The most common use for database programming is in stored procedures and stored functions. Like triggers, stored procedures and functions are durable; they remain in the database after you create them until someone drops them. Creating a stored procedure or function is like creating a table. It is a DDL statement, so it is effectively committed immediately when you do it, and you do not need an explicit COMMIT to make the change permanent and visible to everyone.

Let's continue with the examples in the text, first with a simple procedure that increases the discount by five percent for every product where the quantity on hand is at least twice the minimum quantity.

T-SQL Version of prc_prod_discount Procedure

Note that this procedure has no input and no output parameters. This is intentional, to make this example as simple as possible.

Note the word “AS.”

This could also be “IS.” There is no difference in meaning between IS and AS, but you have to use one of the two. Choose whichever makes the most sense for the procedure that you are writing. The following figure from the text illustrates a test of this procedure.

Reality Check

Discounting is considerably more complex than this trivial example. Most of the products in the product table above would be discounted by this procedure immediately after a minimum size shipment was received, because p_min_reorder is more than twice p_min.

To execute this procedure, we use this command:

```
EXEC prc_prod_discount;
```

Note the line *EXEC PRC_PROD_DISCOUNT*. EXEC is the short form of EXECUTE. This is how you invoke a stored procedure from SQL.

The following screenshot shows the values in the PRODUCT table before the procedure is executed:

And this screenshot shows the values after the procedure is executed:

Procedures with Parameters

In the next example we explore a procedure with parameters and a WHILE loop.

T-SQL Version of PRC_PROD_DISCOUNT Stored Procedure with Parameters

Note the symbol WPI (@wpi in the T-SQL version). This is the formal name of the parameter through which the WPI parameter is passed into the procedure. The WPI parameter is an input parameter.

Note the symbol IN. This could have been *IN*, *OUT*, or *IN OUT*. This required property of each parameter tells Oracle whether to pass the value of the parameter into the procedure, out of the procedure, or both. For complexity control and because parameter passing costs some CPU time it is best to minimize the number of parameters and to only use IN OUT when you need it. About 90% of parameters are IN parameters.

Note that the procedure checks the validity of its input parameter. Stored procedures should always check their input parameters when it is feasible to do so. There is a design problem in what the stored procedure does when an invalid WPI value is detected. What it does is print an error message and return. The code that called this stored procedure won't have a hint that anything was wrong, and will go merrily on its way assuming that the discount has been implemented, which it has not. The procedure should throw an exception or return a result code to indicate to the caller that something is wrong. We will get to exceptions a little later in this lecture.

Notice also that there is no check that P_DISCOUNT, which is the fraction of the discount, does not exceed one. This is a functional error in this procedure. I don't want to harp on this, but it is really important that stored procedures take full responsibility for the correctness of the functions that they are responsible for. It is common that industrial-strength stored procedures are 60% or more error checking and recovery code. Stored procedures should normally check the correctness of all of their input parameters, and any other *preconditions* before doing anything.

A precondition is something that should be true before a module is invoked.

Stored Procedure at Work

The following screen from the Rob and Coronel text shows the stored procedure at work. Note that this execution of a stored procedure from the command line is not how stored procedures are normally executed. Stored procedures are normally invoked by applications code or other stored procedures.

T-SQL Example

Executing the PRC_PROD_DISCOUNT stored procedure with parameters:

We now examine the next stored procedure in the text, which adds a customer to the database.

T-SQL Example

The PRC_CUS_ADD stored procedure with additional notes.

Before we look at this, though, we'll create a sequence to auto-increment the customer code, which is the primary key in CUSTOMER. Creating a sequence in SQL Server 2012 is fairly straightforward.

To use the SEQUENCE in the stored procedure, we use the clause `NEXT VALUE FOR <sequence-name>` to replace the input value for the attribute using the sequence, in this case CUS_CODE.

The first thing that we notice is that this procedure takes five input parameters. This is quite normal. The next thing that we notice is that the stored procedure does not check the validity of its input parameters, but depends on the foreign key and other integrity constraints in the customer table to enforce the integrity for this transaction. This is not a good design.

Notice that a string such as 'Hi there' could be passed in as an area code, and the procedure wouldn't complain. In fact, the whole parameter type structure is unsafe. The W_LN and W_FN input parameters should be type checked to have no more than about 30 characters. The area code should be a NUMERIC(3,0) or INT. There should be a country code input parameter; this is a design bug in the schema, and not only in this procedure. Real customers also have addresses, both snail mail and email, and these should be provided as additional parameters to the procedure that creates a customer.

Notice that in the example above the attempt to execute the procedure passing nulls to the last three parameters cause an error, because the cus_areacode column in the customer table in the teacher schema is NOT NULL.

Notice also the way that this error message is printed, with double quotes around the schema name, table name, and column name. It is a good practice to double quote identifiers when creating tables or accessing or using them via tools. Oracle and other ANSI databases will take a double quoted string as an identifier for a table or column, even if the string contains spaces or other characters, or if the identifier is a reserved word. Thus the double quotes serve as a kind of safety net that improves the likelihood that your create statements will work and that your output will be understood by users who see things like “discount price” as a column name.

The following screenshot shows the execution of this procedure:

Reality Check

The authors are simplifying the examples to make it easier for you to follow them. Real production-quality examples are about twice as complex than these textbook examples. The main additions are input and precondition checking and exception handling code, which is typically about half of a production stored procedure.

Test Yourself

Select all that are true about stored procedures:

Stored procedures can take parameters.

This is true.

Stored procedures can return a value.

This is true. Stored procedures can return values through IN or IN OUT parameters.

Stored procedure requires a COMMIT statement before the stored procedure is durably stored in the database.

This is false. Because a stored procedure is created using DDL, it becomes permanent immediately, without an explicit COMMIT command.

Stored Functions

Stored functions are almost like stored procedures. The only differences are that functions return a value through their function name, so that you have to call functions in a context where the function value makes syntactic sense, such as in a select list or on the right-hand side of an assignment statement. The most important parts of the syntax for creating a function is:

```
CREATE [OR REPLACE] FUNCTION [schema.]<function name>[(<parameter
list>)] RETURN <return data type>
[AUTHID DEFINER | CURRENT_USER] [DETERMINISTIC]
```

```
[PARALLEL ENABLE :] [PIPELINED]
IS

<declarations> BEGIN
<executable statements> [EXCEPTION
<exception handler statements>] END [ <function name> ];
```

T-SQL Create Function Syntax

```
CREATE FUNCTION [ schema_name. ] function_name
[ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type [ =
default ] [ READONLY ] }
[ ,...n ]
]
)
RETURNS return_data_type
[ WITH <function_option> [ ,...n ] ] BEGIN
function_body
RETURN scalar_expression
END
[ ; ]
```

I want to explain this function definition syntax a bit, in case you are not familiar with it.

- Words in all capitals represent themselves, though you can use any casing that you wish, and Oracle won't care. For example, your function declarations must begin with FUNCTION, function, Function, or some other casing of the word “function.”
- Text inside of angle brackets, such as “<function name>”, describes the thing that it represents, in this case the name of the function that is being created.
- Items inside of square brackets, such as “[schema.]”, are optional. For example if you have the CREATE ANY FUNCTION privilege you can create a function for Scott by saying “CREATE FUNCTION scott.foo :”.
- Lists of items separated by the vertical bar character (“|”) indicate that the syntax includes one of the items in the list. For example after the RETURN clause you can say “AUTHID DEFINER” or “CURRENT_USER” or neither (because it is in square brackets), but not both. We will explain what this means on the next page.

The Meaning of Clauses

The following table describes what the clauses mean. Most of these clauses also apply to stored procedures, but we didn't tell you about them when we were introducing stored procedures, because it might have been too much to take in at one time.

Clause	Meaning
CREATE FUNCTION	This just signals that you are creating a function. If you use the OR REPLACE syntax then the new function definition will replace any existing function definition. Choose the function name carefully so that it is unique and easy to understand.
[(<argument list>)]	These are the optional parameters. Functions, such as the built-in function SYSDATE, can take no arguments, in which case they can't have parentheses after their names when they are declared or invoked. The <argument list> is a comma-separated list of triples, each of which consists of the name of the parameter, one of IN, OUT, or IN OUT, and the data type.
return data type	This is the data type of the value returned in the function name.
AUTHID clause	This is an advanced security feature which determines if the function will execute with the privileges of the user who created the function or the user who is executing the function.
DETERMINISTIC clause	This tells the optimizer whether it can try to use a cached value of the function result if none of its inputs change, or whether it must call the function each time, for example if the result depends on time or external inputs.
PARALLEL_ENABLE clause	This tells the optimizer that it can try to run this function in parallel with other parts of its invoking SELECT.
PIPELINED clause	Tells the optimizer to return the results of this function iteratively using the PIPE ROW command
Declarations	If present this is a list of declaration statements for the local variables that have as their lexical scope and lifetime that of the function.
Executable statements	A PL/SQL function must have at least one executable statement.
Exception handler statements	Each of these statements consists of WHEN followed by an exception name, followed by a PL/SQL block that handles that exception.
End descriptor	Between the END and the final semicolon you can (and should) place the name of the function. This tells the reader that this is the end of the function declaration, and distinguishes this END from those in the body of the function. It is a good practice to include the end descriptor in all PL/SQL procedure and function declarations, particularly when they are more than one page long.

The DETERMINISTIC, PARALLEL ENABLE, and PIPELINED clauses are more advanced. The AUTHID clause is something that you should understand. Oracle lets us choose which privileges a stored procedure runs under.

By default a stored procedure or function runs with the privileges of the user who created it. This is usually what you want. For example a stored procedure that implements a business transaction as part of a transactional application programming interface should run with the privileges of the owner of the database objects, so that the procedure or function can access and change the tables, views, and other components of the database itself.

Consider the following simple stored function:

T-SQL Version of NUM_EMP Function

This simple function just runs a count of the number of employees in the EMP table. Note that I followed good practice and gave the end the name of the function.

Note the odd select statement, where I selected from the table DUAL. DUAL is Oracle's dummy table. We have to provide Oracle a table name in SELECT statements because Oracle follows ANSI standards, so SELECT statements must include a table. Note that the function is DETERMINISTIC, which means that the optimizer can choose to cache the result.

Let's test this.

T-SQL Execution of NUM_EMP Function

To execute the function in T-SQL, simple select it: select dbo.num_emp(); The response will look something like this:

You notice that our num_emp function returns the correct new value of 15 rather than the old value of 14 from the last time that we ran num_emp. The DBMS software is keeping track of what num_emp depends upon, and re-executes num_emp when necessary rather than using the obsolete cached value. As we will see in the last lecture of week 2, this is similar to the dependency tracking that databases do to determine which result sets in their SQL caches are still valid.

After this has been saved, the “Object Explorer” now displays a new “Scalar-valued Function” called *dbo.num_emp*.

Term Project Note

Please be aware there is a major Term Project in this course worth a large part of your overall grade and I recommend that you begin work on the project now. During this first week, you should be defining potential projects at a conceptual level and discussing them with your facilitator. Click here for Term Project Details.

To help you check your understanding of the material, I have a prepared a short set of review questions with answers - database programming review questions and answers.

Boston University Metropolitan College